BrightIntegrator™

User's Manual

Version 4.0.0

March 2010

Tables of Contents

1.0 Terms and Abbreviations

BrightBuilder : Bright Software's mobile application designer.

BrightForms : Bright Software's form executing engine.

BrightIntegrator : Bright Software's integration engine that allows data exchange mechanisms to and from various data sources.

BrightServer : Bright Software's mobile application server running on a J2EE compliant application server.

Field User : An employee who works outside of the employer business premises most of his or her time. S/he communicates with the company's computer system remotely over either a mobile network (GSM, GPRS, 3G) or a fixed landline. It is sometimes referred as "*mobile worker*".

JDBC (Java Database Connectivity) **:** A Java API that enables Java programs to execute SQL statements. It allows Java programs to interact with any SQL-compliant database.

MOM : Message Oriented Middleware

Mobile Solution : A set of computer software and hardware that are used to automate data exchange process for the mobile field users in industries such as service, sales etc. The mobile solutions are designed and commissioned by the systems integrators.

ODBC (Open Database Connectivity) : A standard database access method.

PAS : **P**ublish **A**nd **S**ubscribe

P&S : **P**ublish **&** **S**ubscribe

Push : Refers to the data exchange that is initiated and controlled by the server.

Synchronisation : Refers to the data exchange that is initiated and controlled by the client side.

1.0 Introduction

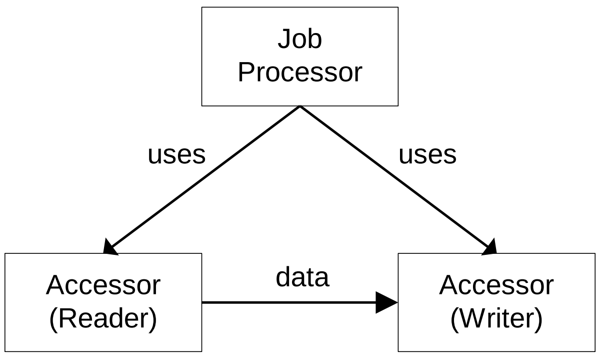
BrightIntegrator™ is Bright Software's integration engine that allows data exchange mechanisms to and from various data sources. These data sources include simple flat ASCII text files (fixed and comma separated values), JDBC (including ODBC via JDBC-ODBC bridge), Pronto via PIE interface, BrightServer™, BrightForms™ field clients (only as a destination), email servers (currently only as a destination), web services etc.

The user can configure BrightIntegrator™ to exchange data in either direction between the data sources; and can also be used to push data to the remote field users using BrightForms™.

The data exchange is defined with an XML configuration file. The user configures a set of jobs consisted of one or more tasks to be executed by the BrightIntegrator™ engine .

1.1 The Job Processor

The central component of the BrightIntegrator™ engine is the Job Processor (JP). JP is mainly responsible for managing the data flow from a source to a destination. The Job Processor can also handle the optional calculation of the difference between source and destination data (sometimes referred as *task data* throughout this document), as well as the optional grouping of the source data. JP manages the data transactions across the whole job. It drives the data accessors to co-ordinate the data read and write. The following image shows the simple BrightIntegrator architecture with the Job



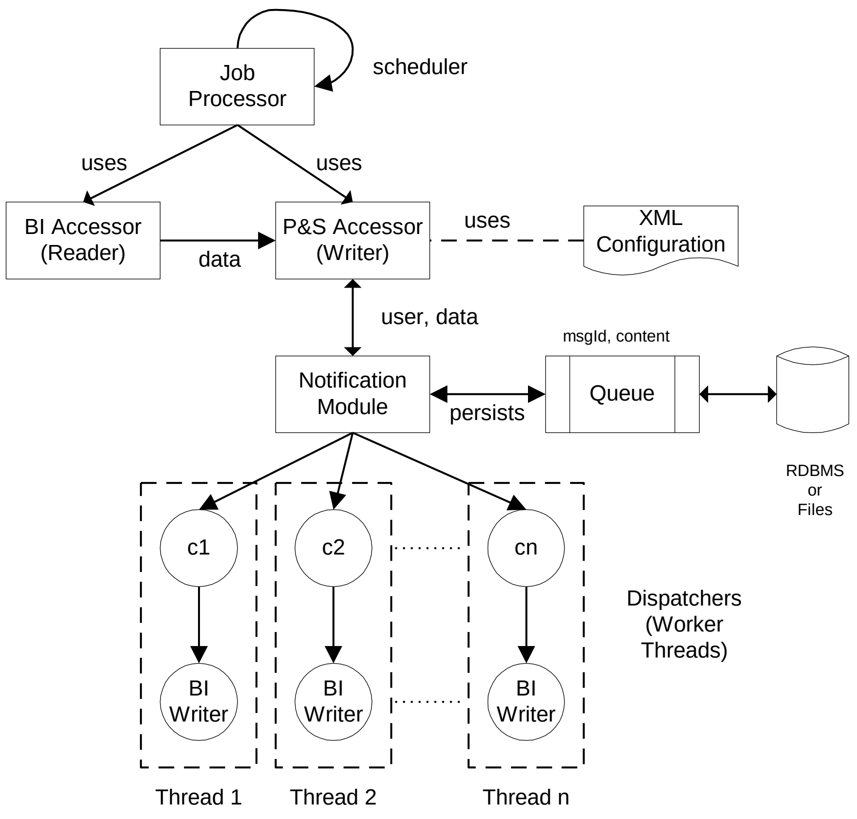
BrightIntegrator Architecture

A data accessor is a component that knows how to read from and write to a particular data source (BrightServer™, JDBC data source, file etc.). Accessors are configured in a task to operate in *read* or *write* mode. A series of configured tasks defines a job. In simple terms, for configured tasks in a job, the JP reads data from a source data set using an accessor operating in *reader* mode, and writes the data read to a destination data set using an accessor operating in *writer* mode.

1.2 Push Module

Sending (pushing) a dataset to a destination as it becomes available is one of the important requirements of any backend integration. Especially this may become critically important, for instance, for organisations who need to push a job or service requests to the remote field users as they receive them from their customers.

To address this requirement a special writer accessor is used : Push Accessor. The push writer accessor incorporates the Notification module that allows BrightIntegrator to send data to the field users using the Publish and Subscribe modules. The Publish and Subscribe modules is a rules based system that knows what data to send and where to send. This is basically a push module that allows the server to dispatch data automatically to the field users. The following image shows the Push module architecture:



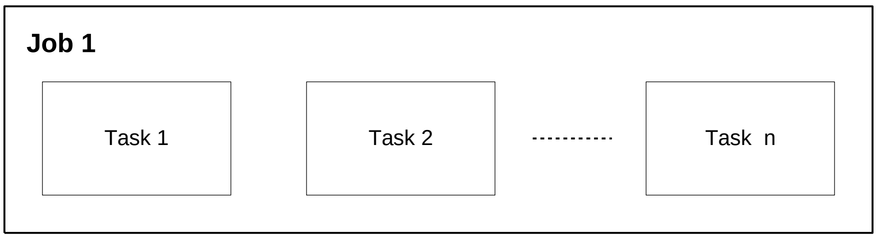
BrightIntegrator Push Module Architecture

This functionality will be discussed in more detail in Section 5 - Push Module.

2.0 Jobs and Tasks

BrightIntegrator™ executes a job or a series of jobs defined in the XML configuration file. The name of the configuration file can be passed to BrightIntegrator™ as one of the command line arguments.

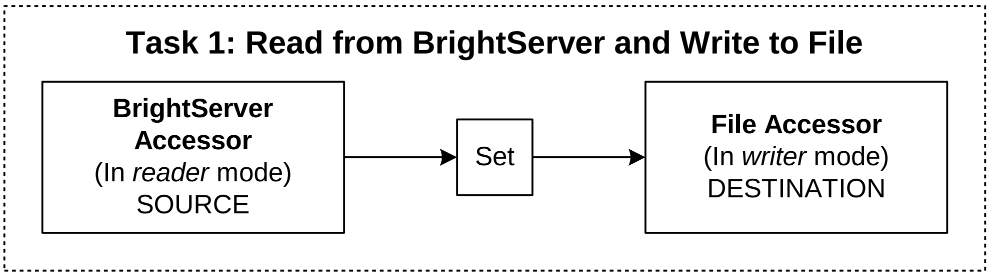
A job then consists of tasks. A task is basically a read action from a data source in order to obtain data, and then a write action to write that data to the destination. The Job Processor creates and maintains the lifetimes of each task configured in a job. The following diagram depicts a job.



The Job Processor will create the necessary accessors to complete the read from the source and subsequent write to the destination. The source accessor is used in the ***reader*** mode, and the destination accessor is operated in the ***writer*** mode.

The accessor configured as the source (i.e. the data reader), reads the data and passes the data as a set to the Job Processor. The Job Processor then writes the set received from the reader to the writer.

A data reader (source accessor) may be able to read the data in chunks, rather then in a single big read. The Job Processor can query the reader accessor and, if the chunking is configured for the data set and the reader accessor supports data chunking, then the data is read from the source in iterations and written to the destination in blocks of the specified size. The size of the block is configurable for each data set defined (See section 3.1 Data Iteration and Chunking for further details).



The data read from the source is converted into a set. A set is a very important logical entity within BrightIntegrator™. It represents the basic processing unit within BrightIntegrator™. Data arrives from the source, typically as records from the same table, or lines from the same file. The incoming data will be formatted and arranged according to the convention of the data source, and will need to be parsed and converted to Bright Software data types. So finally the set will be vendor neutral and decoupled from the data source, and ready to be processed, and then written to the destination. See section 4.0 Data Mapping for further details about how the data mapping takes place.

When it comes to writing the data to the destination, each set is matched by name from its source to its destination. Each set contains a mapping that provides a name for each field. Each field is also matched by name from within its source set to its destination set. In this way, when transferring data, the set names, and the names of each field within the sets must correspond between source and destination.

If the push accessor is used in a task as writer (destination) accessor, then the task via the push module being used can process the incoming data and deposit the messages to the Notification Module. Please note that the push accessor has its own configuration file that contains the necessary configuration aspects of the push module. See Section 5.0 Push Module for further details about the accessor writer.

2.1 Transaction Support

A task may be executed in the context of a transaction. JP manages transactions by tightly controlling the data writer for each task. Recall that the data writer is the accessor operating in ***writer*** mode for that task.

JP indicates the start of a transaction to the data writer, then the actual task is carried out, and finally, if successful, JP causes the data writer to actually commit the data. In the event of a failure, JP asks the writer to roll back the transaction data, to the same state as before the start of the transaction.

For configuration purposes, each task entry in the job definition can specify the start and/or end point of a transaction. These flags are as follows.

Flag	Description
bt	Begin Transaction
et	End Transaction

The Job Processor starts a transaction if the **bt** flag is set to “1”, and completes the transaction if the **et** flag is set to “1”.

In this way, a group of tasks can be defined to be in a transaction together. For instance, to have three tasks A, B and C performed as a transaction. The flags would be

A **bt**=1 **et**=0
B **bt**=0 **et**=0
C **bt**=0 **et** =1

If a task fails in transacted group of tasks, then all the tasks in the transacted group are deemed to have failed. In this case, all data from all of the tasks will be rolled back to the same state as before the first task commenced.

2.1.1 Auto-Commit

If both the transaction flags are turned off (i.e. **bt**=”0” and **et**=”0”), and the task is outside of any transaction group, then the task is said to be in “***auto-commit***” mode.

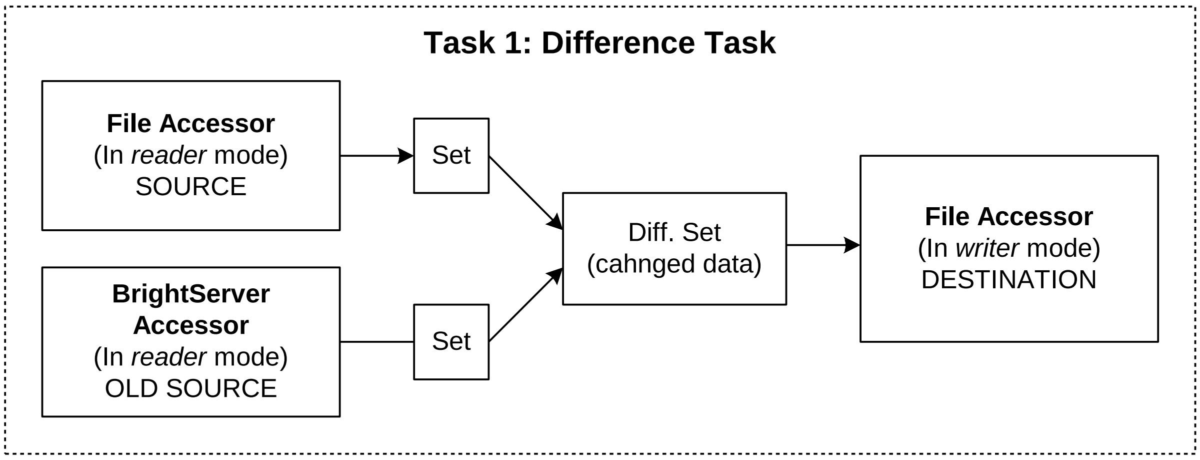
This means that data will be actually written, or committed, as soon as it is received by the data writer.

2.1.2 Auto-Commit and BrightServer™

It is recommended to use auto-commit when writing to BrightServer™, whenever possible, especially when importing large amounts of data. In general, transactions should only be used when dealing with data integrity across multiple tasks. Since transactions require that all written data be temporarily stored until the commit occurs, large amounts of memory are required. In addition, each transaction has a corresponding timeout on the application server, which will elapse if the amount of data is too great.

2.2 Calculating the Difference

The Job Processor may be configured to calculate the difference between two data sources being read. This feature can minimise the load and processing on the destination, hence reducing the overall write time. For instance, if a large inventory file needs to be loaded into the BrightServer™ database, calculating the difference (i.e. what has changed since the last data load), and just importing the changed data will reduce the load and the processing time for the subsequent inventory updates.


Task 1: Difference Task
File Accessor (In reader mode) SOURCE
Set
BrightServer Accessor (In reader mode) OLD SOURCE
Set
Diff. Set (cahnged data)
File Accessor (In writer mode) DESTINATION

The task definition has a parameter that the user can use to define the previous old data set. If the old data source is defined, then the Job Processor will read from both old and new data sources, then calculate the difference, and finally write the difference data onward to the destination.

Data that has been read is stored in sets. Sets are used so that data from disparate sources can be compared. For example, using sets, it is possible to compare data from a file and a server table. The difference between the old set and the new set is processed by matching sets of the same name, and comparing the field values of the primary keys. (Primary keys are specified in the data mapping). If the primary keys match, then the rest of the data in the record are compared. If some data has changed, then the newer record is appended to the difference data, and a flag is added to the record, signifying that the record has “changed”. If the data is identical, then nothing is appended to the difference data.

If a new record with the primary key values is not found in the old data set, then the record is appended to the difference data, and a flag is added, signifying that the records has been “added”. Vice versa, old records with no matching new record are said to have been “deleted”.

The result itself is another set, and each record will have a difference status flag, “added”, “changed”, or “deleted”. The result set is written to the destination, which must be able to understand the difference status of each record, and act accordingly. Types of data sets that are able to consume difference data include BrightServer™ and JDBC.

2.3 Grouping Data

The Job Processor may be configured to sort the data that has been read, into logically related groups of data. The data that has been read will be stored in sets. By grouping the data, these sets of records will be transformed into groups of data.

2.3.1 Group Definition

A group is a set of records from sets which are associated by grouping relationships. A group will contain one record from the parent set and all its related records from the child set(s).

The grouping relationship describes how the member records from each set, are related to each other.

For example let us assume that there are two data sets that have been read. These two sets are ORDER_HEADER, containing records for new sales orders, and ORDER_ITEM, containing records for the items that comprise the new sales orders. Now say that this new sales order data needs to be sent to a destination data writer that only understands orders, rather than sets or tables. This is a case we would need to transform the two sets into groups. In this case each group will be an individual sales order. Consider the following data for tables ORDER_HEADER, and ORDER_ITEM.

ORDER_HEADER

ORDER_ID	ORDER_DATE	CUST_ID
100	1/1/2004	1234
101	2//1/2004	5678

ORDER_ITEM

ORDER_ID	ITEM_NO	PRODUCT_ID	QTY
100	1	PRD1	10
100	2	PRD2	15
101	1	PRD2	5
101	2	PRD15	10
101	3	PRD2	1

From this example data, we would have two groups as follows:

Group 1

ORDER_HEADER

ORDER_ID	ORDER_DATE	CUST_ID
100	1/1/2004	1234

ORDER_ITEM

ORDER_ID	ITEM_NO	PRODUCT_ID	QTY
100	1	PRD1	10
100	2	PRD2	15

Group 2

ORDER_HEADER

ORDER_ID	ORDER_DATE	CUST_ID
101	2//1/2004	5678

ORDER_ITEM

ORDER_ID	ITEM_NO	PRODUCT_ID	QTY
101	1	PRD2	5
101	2	PRD15	10
101	3	PRD2	1

The way that the grouping process works, is that it takes the parent set (ORDER_HEADER in the case above) and creates a group for each record. Thus each group is initially created, and has its first member record. Next, the process takes each child set, (ORDER_ITEM is the lone child set in the case above) and with the grouping relationship in mind, places each child set record into the group that it matches, according to the relationship key field. In the case above, the ORDER_ID is the key field that relates the ORDER_HEADER set to the ORDER_ITEM set.

Groups are defined in the Tasks element of the configuration file. See Section 7.2.1 for more details on groups.

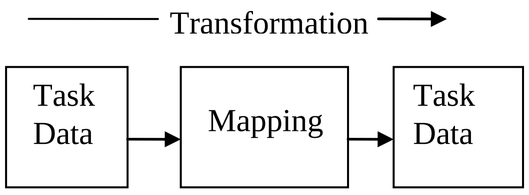
It is also possible for a single set to be "self-grouped". This one set will have each of its records placed into a single group by themselves. The result is that there are as many resulting groups as there were records in the single set. It is like an order header set being grouped so that each order header record appears in its own group, but then there are zero order items to be grouped into the order groups.

2.4 Transforming Data

Sometimes the data read from a source needs to be slightly manipulated before it is written to its destination. For instance a new field may be needed in the data to identify its source before written to its destination. This would require an additional field to be created in the task data. Using *transformations*, a task data (the data read from a source) can be modified to have an extra field in addition to the fields read from the source.

Another example would be the need to merge the fields from various records in different sets into a single record. For example, an order items may need to be combined with the fields from their order header data.

The JP (Job Processor) can be configured to transform a task data (grouped or not) defined as per the *transformation* elements in a task definition. The JP simply will process each transformation defined for each set and transform those sets in the task data as per the mapping and other configuration options specified in the transformation definition.



Transformations are executed by the JP after finding a difference (if an old source is configured in the task definition) and grouping the data (if the grouping options are specified).

The transformations are defined for each set. Each set transformation can be configured to transform the set itself, or transform the set into another set by specifying the optional output set name. The transformed task data will then contain the new sets and the existing sets prior to the transformations.

3.0 Data Sets

Each task basically reads from a source and writes to a destination. A task is configured to do so by naming data sets to act as the source and destination. Data sets define the locations where the actual data resides.

Data sets come in a variety of types. These include ASCII files, BrightServer™ tables, JDBC data sources, and more.

See section 7.3 Data Sets Configuration for the available data sets and for details on how to configure them.

3.1 Data Iteration and Chunking

The simplest way to think of a task being performed is that all the data is read from the source, and then that data is written to the destination. However, in the case of a large amount of data to be transferred, this simple approach will cause problems, in the form of unresponsiveness and timeouts.

A better way to deal with large data transfers is to use the data iteration feature of BrightIntegrator™. Data iteration will read a "chunk" of data from the source, and then write the chunk to the destination, and this is repeated until all of the data has been processed. In this way, there are write operation interlaced within the read operations.

A task will be performed using data iteration if the source has a data-set limit defined. The size of the each chunk for the iterations will be up to the defined limit. There are two exceptions to this however. If a difference is to be processed, or if grouping has been configured for the task, then no data iteration is possible. This is due to the fact that in both cases, all of the data must be read and processed, before any data can be written.

REMEMBER: You cannot use data iteration and chunking when processing differences and grouped data.

You will know that BrightIntegrator is iterating and chunking the records when it displays "INFO Iterating data from BrightServerBudgetTable to BudgetFile" on the output screen.

3.1.1 Data Iteration and BrightServer™

When using data iteration to write to BrightServer™, data chunks of multiple records will arrive at the BrightServer™ to be processed. Consider the case where one of those records contained some invalid data, such as a string that was too long for its field, and as such, would not be able to be processed. The result would be that the entire chunk will fail to be processed.

Given this scenario, if the task is in auto-commit mode, then BrightIntegrator™ will retry to submit each record in the failed chunk, one at a time. In this way, the good data before the bad record will still get through, and the bad record will be isolated. This then allows a detailed error regarding the bad record to be returned to the user. This useful feature is only possible if auto-commit is enabled for the task. This is another reason why using auto-commit is strongly recommended when writing to BrightServer™.

IMPORTANT : Use auto-commit especially when writing large amount of data to BrightServer.

4.0 Data Mapping

Since the BrightIntegrator™ allows us to read data from various disparate data sets, a need arises for mapping between data types. The problem is that the data types on the source data set might not be understood by the destination data set. The solution is to map all external data types to common internal data types. In this way, all data that is read is interpreted by the data reader and given to JP as internal types. Then any optional difference or grouping is carried out. Finally, the data writer receives its data as internal types, which it translates to the external types for writing.

In this way, each data set uses its own mapping to translate data to and from the internal data types. The mapping is also used to specify the primary keys, which are used in the processing of groups, as well as calculating differences between records.

Each data set type has its own way of representing data mappings.

4.1 Data Mapping and BrightServer

IMPORTANT NOTE: The data mapping for the BrightServer data set ***MUST*** follow the ***column order*** of the registered table in BrightServer. It must include all table columns.

The BrightServer reader accessor populates the data set according to the column order defined by the table configuration stored in BrightServer. If the data mapping for the writer data set is ordered the same as the BrightServer data set, the data mappings will be easier to determine by JP. Otherwise, JP will try to map each field at a time.

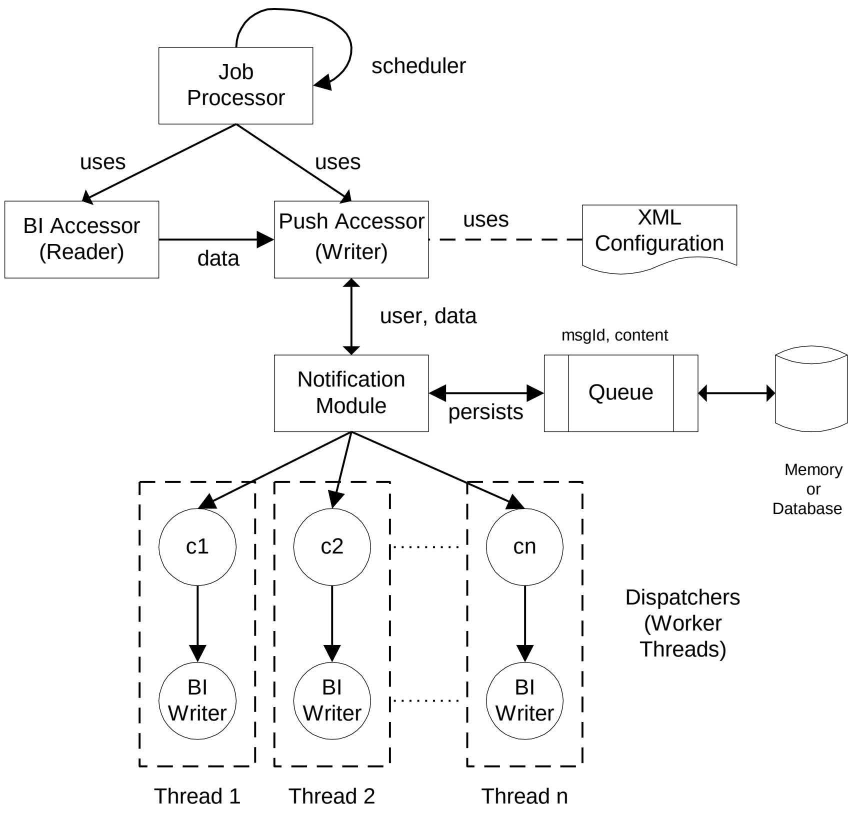
Since the query returns all the fields from the server in the column order of the table defined, the Query <outputfields/> element does not need to be specified in the configuration file.

5.0 Push Module

By definition, the data push means sending data automatically to a subscriber based on a set time or a certain criteria of circumstances defined in a schedule. In mobile solutions, this service allows supervisors to send data to their field users, it is more effective in terms of the time sensitivity nature of the data that needs to be dispatched to the field as soon as it can be and more efficient in terms of the data transfer optimisation and network usage.

The Push Module in BrightIntegrator has two modules, the ***Push Accessor*** and the ***Push Notification Module***. *Publish Module* is the rule based system that knows what to send and where to send, while it is the responsibility of the *Push Notification Module* to find the destination of the data and send it.

5.1 Architecture



BrightIntegrator Push Mechanism Architecture

The Push Module is implemented as a BrightIntegrator accessor and will pass the user and message details to the Notification Module. The Notification Module is hosted within the Job Processor as a sperate component. It is instantiated via the first invocation of the Push Accessor. From then onwards, the push accessor writer object processes the incoming task data and deposits the messages to the Notification Module.

The Notification Module then handles the data distribution using the Dispatcher objects. The messages are persisted in a queue for disaster recovery and resumption of message delivery purposes. The default message store type is memory. That means that the message queue is not saved between restarts of BrightIntegrator.

The dispatcher objects use native BrightIntegrator accessors to send data to its destination. This provides a consistent way of reading data from a source and writing it to a destination. All the accessors that support the ***writer*** functionality can be used in a dispatcher configuration to push data. For instance, it is possible to read a file content and send it to a user via an email accessor.

The new push accessor has its own configuration file that contains all the necessary configuration aspects including subscription and publication rules, the dispatcher configuration and their delivery attributes.

A scheduler feature has also been implemented on top of the existing Job Processor module. The scheduler component can trigger the data reads from the source configured based on the job's schedule defined. You can use simple scheduling based on an interval or use a more complex Cron-trigger based scheduler. Read more about the Cron triggers in Appendix B.

5.2 Push Module XML Configuration File

The following is the layout of the Push Module configuration file:

```
<push-task version="1.0" def-version="1">
 <publications>
   <publication name="PublishJobs" set="jobs">
      <field>tech_id</field>
      <field>state_code</field>

       <!-- more field's -->
       <!-- ... -->
   </publication>
   <!-- more publication's -->
</publications>

 <subscriptions>
  <!-- A subscription must reference a publication, optional:default-dispatcher-->

  <subscription publication="PublishJobs" default-dispatcher="BFNotify">
   <subscriber type="constant">
    <subscriber-values>
     <!-- for type=constant, the values are the actual data values -->

     <value name="tech_id" type="int">100</value>
     <value name="state_code" type="string">NSW</value>

     <!-- more values, corresponding to the named fields in the referenced
publication-->
    </subscriber-values>

    <dispatcher-values dispatcher="BFNotify">
    <!-- dispatcher-values may use default value from its subscription -->

     <value name="ip" type="string">127.0.0.1</value>

     <!-- more values, corresponding to parameters of the dispatcher being used-->
     </dispatcher-values>
   </subscriber>
   <subscriber type="file" file="testData/techmap.csv">
    <subscriber-values>
     <!-- For type=file, the values are indexing the file columns -->

     <value name="tech_id" type="int">1</value>
     <value name="state_code" type="int">2</value>

     <!-- more subscriber-value's -->
    </subscriber-values>

    <dispatcher-values dispatcher="BFNotify">
    <!-- dispatcher-values may use default value from its subscription-->
     <value name="ip" type="int">3</value>
     <!-- more values, corresponding to parameters of the dispatcher being used-->

    </dispatcher-values>
   </subscriber>
   <!-- more subscriber's -->
  </subscription>
  <!-- more subscription's -->
</subscriptions>

 <dispatchers>
  <!--Dispatcher "destination" - name of the dataset to be used to send data-->
  <dispatcher name="BFNotify" destination="BF">
   <attribs>
    <value name="retries" type="int">3</value>
    <!-- more "value"s -->
   </attribs>
   <escalations>
    <!-- executed on-success (0..many) -->
    <escalation execute="on-success">
     <!-- destination writer -->
     <destination>Email</destination>
     <!-- ANDed field values in the header record -->
     <condition/>
     <!-- New field values in the header record. Will replace existing ones -->
      <write-values/>
    </escalation>

    <!-- executed on-failure (0..many) -->
    <escalation execute="on-failure">
     <destination>BSAccessor</destination>
     <condition>
     <value name="STATUS" type="int">0</field>
     </condition>
     <write-values>
     <value name="FAILED_FLAG" type="int">1</field>
     </write-values>
    </escalation>

    <!-- executed always (0..many) -->
    <escalation execute="always">
     <!-- destination writer -->
     <destination>Email</destination>
     <!-- ANDed field values in the header record -->
     <condition/>
     <write-values/>
    </escalation>
   </escalations>
  </dispatcher>

  <!-- more dispatcher's -->
</dispatchers>

 <notifier>
  <attribs>
```

```
   <value name="max-retries" type="int">3</value>
   <value name="retry-interval" type="int">1800</value>
   <value name="delete-failed-jobs" type="boolean">yes</value>
  </attribs>

  <message-store type="database">
   <value name="url" type="string">127.0.0.1</value>
   <value name="jdbc-driver" type="string">drivername</value>
   <value name="username" type="string">bsuser</value>
   <value name="password" type="string">bspassword</value>
  </message-store>
 </notifier>

</push-task>
```

5.3 Message States And Retries

Messages to be sent are persisted in a queue called the message store. This will contain details of the failed message i.e. the data to be sent or where to send it so that it can be resent to the destination.

There are two types of message stores that can be configured in BrightIntegrator: A ***memory*** based message queue and a ***relation database*** based message queue. The memory based message store is a temporary queue and lost between the restarts of BrightIntegrator.

By default, if there is no message store defined, the ***memory*** message store type is used. When a database message store is used, then BrightIntegrator needs a table with the following name and structure created.

SQL scripts are distributed in the "resources" directory in BrightIntegrator distribution.

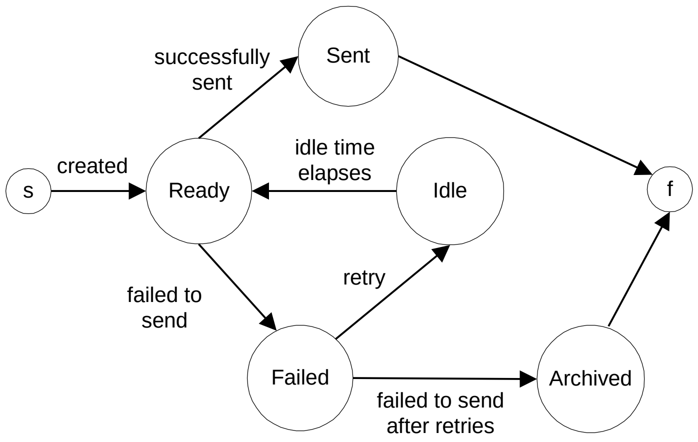
BS__MSG_STORE

Column Name	Type	Description
MSG_ID	Integer	Unique message id
DT_ACTION	DateTime	Date and time of the last action on the message (created, or updated)
MSG_STATUS	Integer	Message state, see table and diagram below for integer values and descriptions; and the corresponding state diagram.
RETRIES	Integer	No of current retries for the message.
CONTENT	Image/Blob	This column is the binary blob column where the serialised message object is stored.

Message Status and its corresponding descriptions are given in the table below.

Message Status	Description
0	Ready
1	Idle
2	Sent
3	Failed
4	Archived

The message status state machine is as follows.



When a message is created, it will be in the “Ready” state and persisted to the message store. When the message is sent to its destination successfully, then it will be in the “Sent” state, and will also be removed from the message store.

If the message could not be sent, then it will be put into the “Failed” state. If the “retries” option is configured (i.e. > 0), then the Push Module will put the message into the “Idle” state for the duration of the configured “idle time”.

When the idle time elapses, then the message will again be in the “Ready” state and the Push Module will try to send it to its destination. If the message could not be sent after the configured “retries”, then it will be put into “Archived” state in the message store.

“Archived” messages can be configured to be deleted automatically from the message store if the “delete failed messages” option is configured in the Push Module.

The “retries”, “idle time” and “deleted failed message” are attributes of the Notification Module and can be configured from the “**notifier**” section of the Push Module XML configuration file using the <max-retries>, <retry-interval> and <delete-failed-jobs> elements. See the Push Module Configuration in Section 8 for further details on the “**notifier**” elements.

5.4 Message Escalation

The Push Module can also be configured to escalate messages. These escalation actions can be executed 1) on-failure, 2) on-success or 3) always.

If any escalation is configured, the Push Module will send the failed or successful message to the configured destination. The Push Module will also be able to escalate message based on the message content (field values in the task data of the message) to different destinations. The escalation configuration will also allow the designer to overwrite the existing message values, which, in turn, can be used to provide a feedback back to the source of the message.

Escalation destination points can be one of the available BrightIntegrator accessors. That will mean that the message success or failure can be notified back to BrightServer, or a JDBC accessible database table, or an email to system administrator etc.

5.5 Push Module Accessors

There are a couple of data-sets available for the Push Module, namely:

- BrightForms – defines the configuration for BrightForms client that the message will be sent to or a sync-rule will be executed upon.
- Push – defines the Push Module configuration file.
- Email – defines the email configuration for the escalation messages.

The usual accessors such as BrightServer, File and JDBC tables can also be used as a destination data-set for the push module.

Each of these accessors has different attributes that can be overwritten based on the subscriber details. For example, IP addresses of each BrightForms device for each field users.

These accessors will be discussed in detail in the Data Sets Configuration section.

5.6 Publications and Subscriptions

Publications and subscriptions are always defined within the context of the set data, therefore, each publication and subscription is associated with a specific set data. Publications are defined by the query or set data of an accessor. The Push Module can have many publications and subscriptions.

Before you can send data to a subscriber, you must first create a publication for that data and also create the subscriptions (who that data is to be sent to). Each publication can contain many different fields to limit the records to send to each subscribers, this allows you to filter the task data based from the set data you defined in the Publication element of the Push configuration file. Each subscription refers to (subscribes to) a specific publication thus should also contain the same number of fields that the publication refers to.

5.7 The Synchronisation Engine and the Push Module

BrightIntegrator works hand in hand with the normal synchronisation engine of BrightSoftware platform. If using the BrightForms accessor, the BrightBuilder developer should incorporate the normal synchronisation process in mobile application. If using background synchronisation, this should also be included in the application program.

6.0 How to Install and Run BrightIntegrator™

To install BrightIntegrator, simply uncompress the contents of BrightIntegrator_2_X_X.tgz to a hard disk directory, i.e. c:\brightintegrator.

Follow the steps below to run BrightIntegrator:

1. Open a command prompt window
2. Change directory to BrightIntegrator installation directory, i.e. c:\brightintegrator
3. Type the command **run** in the command prompt.

This will execute BrightIntegrator™ using the default config file, which is **config.xml** in the conf directory. An alternate config file can be specified on the command line using –c.

For example, run –c D:/MyConfig.xml

The other command line option is –n, which means "no retries". By default, if BrightIntegrator™ experiences an error during a job, then the next time it runs that job, it will retry by starting at the last task in that job that was successful. By specifying the –n command line option, BrightIntegrator™ will start each job from the first configured task instead.

The location and name of the last-run file (see Chapter 9) can be set by using [-l|--lastrun LASTRUN_FILE_NAME] option. The default location of this file is the ***conf*** directory.

Important: Ensure that the tables required in the BrightServer data set has been created and registered in BrightServer through the Management Console.

7.0 A Brief Introduction to XML

This section is designed for users that are unfamiliar with XML documents. If you are familiar with XML basics proceed to next section on configuring BrightIntegrator.

XML stands for E**x**tensible **M**arkup **L**anguage. It is a structural and semantic language, not a formatting language. XML documents form a tree structure, made of elements. Element and attribute names reflect the kind of the element being described.

```
<PERSON ID="p1100" GENDER="M">
  <NAME>
    <FORMAT type=”1”></FORMAT>
    <GIVEN>Judson</GIVEN>
    <SURNAME>McDaniel</SURNAME>
  </NAME>
  <BIRTH>Birthday
    <DATE>21 Feb 1834</DATE>
  </BIRTH>
  <DEATH>
    <DATE>9 Dec 1905</DATE>
  </DEATH>
</PERSON>
```

In this example PERSON is the root element and has several child elements. Every element is opened and closed with a start tag (<PERSON>) and end tag (</PERSON>). An **XML element** is everything from (including) the element's start tag to (including) the element's end tag.

PERSON is the **parent element** of NAME, BIRTH and DEATH. NAME, BIRTH and DEATH are **siblings** because they have the same parent.

An element can have **element** content, **mixed** content, **simple** content, or **empty** content. An element can also have **attributes**.

In the example above, PERSON has **element content**, because it contains other elements. BIRTH has **mixed content** because it contains both text and other elements. DATE has **simple content** (or **text content**) because it contains only text. FORMAT has **empty content**, because it carries no information.

XML elements can have attributes. Attributes offer information about a particular element. In the example above, PERSON has two attributes `ID="p1100" and GENDER="M"`. The **attribute** named ID has the **value** "p1100". The **attribute** named GENDER has the **value** "M".

8.0 How to Configure BrightIntegrator™

Configuration of BrightIntegrator™ is carried out by editing the XML configuration file. The overall structure of BrightIntegrator xml configuration file is as follows:

```
<integrator version="2.0">
   <jobs version="2.0">
      <job name="ExportBarCodesJob">
         <task-entry name="ExportBarCodesTask" bt="1" et="1"/>
         <!--List more task-entry here... -->
      </job>
      <!--List more jobs here... -->
   </jobs>
   <tasks version='2.0'>
      <task name="ImportBarCodesTask" >
         <!--Task Details... -->
      </task>
      <!--List more task here... -->
   </tasks>
   <queries>
      <query name="BarcodeQuery">
         <!--query details... -->
      </query>
      <!--List more queries here... -->
   </queries>
   <data-sets>
      <data-set name="BrightServerBarCodeTable" type="BrightServer">
        <!--data-set details... -->
      </data-set>
      <!--List more data-sets here... -->
   </data-sets>
   <mappings version = "2.0">
      <mapping name="BarcodeQuery" type="query">
        <!--mapping details... -->
      </mapping>
      <!--List more mappings here... -->
   </mappings>
   <schedules>
      <schedule name ="SimpleSchedule" type="simple">
         <value name="interval" type="int">300</value>
      </schedule>
      <schedule name ="CronSchedule" type="cron">
         <value name="cron-expression" type="string">0 0/5 * * * ?</value>
         <!--List more values here... -->
      </schedule>
   <!--List more schedules here... -->
   </schedules>
</integrator>
```

Elements in “integrator”

Element Name	*Description*	*Required*
`jobs`	Defines the jobs to be executed within BrightIntegrator. This consists of one or more tasks elements.	Yes
`tasks`	Describes the task details i.e. source and destination data sets, referred by the jobs elements.	Yes
`queries`	States what data are to be transferred. Used by BrightServer and JDBC data-sets.	Yes
`data-sets`	Data sets define the locations where the actual data resides.	Yes
`mappings`	This element provides a unique, logical name for each data field in the data set.	Yes

The file must contain the following elements: jobs, tasks, queries, data-sets, and mappings. The details of each element are provided in the following sub-sections. Also refer to the default configuration file, ***config.xml***, located in the conf directory of BrightIntegrator installation directory.

8.1 Jobs Configuration

Each job must have a name, and consists of one or more task-entry’s that will be executed in order of appearance. Each task-entry names the task that is to be executed.

```
<job name=”MyJob”>
      <task-entry name="AutoCommitTask" bt="0" et="0"/>
      <task-entry name="TransTask1" bt="1" et="0"/>
      <task-entry name="TransTask2" bt="0" et="0"/>
      <task-entry name="TransTask3" bt="0" et="1"/>
</job>
```

In the above example, the first task to be executed will be an AutoCommitTask, which is said to be in “auto-commit” mode. This task will write its data as soon as it is sent to the data writer. The following three tasks will be executed sequentially, as a transaction.

Attributes of “job”

Attribute	*Description*	*Required*
`name`	The name of the job.	Yes
`schedule`	The name of the scheduler component to be used to trigger the job executions.	No
`continue-on-error`	If specified and set to “yes”, then if a task fails, BrightIntegrator will continue with the next task defined in the job. If the task failed is in a transaction with other tasks, then BrightIntegrator will omit the tasks in the same transaction, and execute the next task that falls outside of the transaction.	No Default set to “no”

Elements in "job"

Element Name	*Description*	*Required*
`task-entry`	The name of the task that will be executed in order of appearance.	Yes

Attributes of "task-entry"

Attribute	*Description*	*Required*
`name`	The name of the task to be executed.	Yes
`bt`	The begin transaction flag. Refer to overview in section 2.1 Transaction Support	No, default is zero
`et`	The end transaction flag. Refer to overview in section 2.1 Transaction Support	No, default is zero

8.2 Tasks Configuration

Each task must have a **name** attribute, a **source** element, and a **destination** element. Here is a sample task definition:

```
<task name="MyTask">
      <source>BarcodeFile</source>
      <destination>BrightServerBarCodeTable</destination>
      <description>
            <![CDATA[Read barcode table records from file]]>
      </description>
</task>
```

Attributes of "task"

Attribute	*Description*	*Required*
`name`	The name of the task. This name will be referred to by the jobs.	Yes

Elements in "task"

Element Name	*Description*	*Required*
`source`	The name of the data-set that will be read from, and provide the data.	Yes
`destination`	The name of the data-set that will be written to, and receive the data.	Yes
`description`	Provides a brief description of the task.	No
`old-source`	The name of the data-set that will be read from, in	No, this element

	addition to the source. Then difference between both sets of data will be written to the destination. Refer to overview in section 2.2 Calculating the Difference.	will trigger the difference processing.
grouping	This element defines the relationships that will be used to associated data into groups. See subsection 7.2.1 Grouping for details.	No, this element will trigger the group processing.

8.2.1 Grouping

A task may declare a grouping element that will trigger the group processing algorithm. This will group associated data records together according to the relationships defined here. Refer to overview in section 2.2 Grouping Data.

Here is a sample task definition that contains grouping:

```
<task name="GroupTask">
      <source>OrdersFile</source>
      <destination>PIEServer</destination>
      <grouping>
            <relationship>
                  <src set="OrderHeader">
                     <key>OID</key>
                  </src>
                  <dst set="OrderItem">
                     <key>OID</key>
                  </dst>
            </relationship>
      </grouping>
</task>
```

The above sample extends the example use case discussed in Section 2.3 Grouping Data. In this case, we have OrderHeader and OrderItem files arriving as the source. We want to group the incoming data into orders, and the way that the data in the two files relates to each other, is via the OID, or Order Identification Number. Each OrderHeader record contains a unique OID. Each OrderItem shows that it belongs to an OrderHeader, by also having an OID.

Now relating this use case to the sample task-grouping element, we see that the grouping element contains a relationship element. The grouping may contain one or more relationships. The exact number of relationships required is equal to the number of sets in the source data, minus one. In this use case we have two sets, OrderHeader and OrderItem. Therefore we must define exactly one relationship.

The relationship element must contain a src (source) and a dst (destination) element. A relationship is one-to-many, from source to destination. Both src and dst must have a name, which corresponds to the name of a set in the incoming data, and a key element, which names the field used to relate the two sets.

Elements in "grouping"

Element Name	***Description***	***Required***

`relationship`	This element defines the relationship keys of the source (parent) and destination (child) sets.	Yes, if grouping is required.

Elements in “relationship”

Element Name	*Description*	*Required*
`src`	The source set to be used in the relationship.	Yes, if grouping is required.
`dst`	The destination set to be used in the relationship.	Yes, if grouping is required.

Attributes of “src”/”dst”

Attribute	*Description*	*Required*
`set`	The name of the set in the incoming data.	Yes, if grouping is required.

Elements in “src”/”dst”

Element Name	*Description*	*Required*
`key`	The name of the field that relates the two sets.	Yes, if grouping is required.

8.2.2 Transformations

A task may declare a <*transformations*> element that may contain a list transformation defined by <*transformation*> elements. This will group associated data records together according to the relationships defined here. Refer to overview in section 2.4 Transforming Data.

Here is a sample task definition that contains transformation:

```
<task name=”TransformTask”>
      <source>InData</source>
      <destination>OutData</destination>
      <transformations>
         <transformation set="OrderHeader" output-set="NewOrderHeader">
             <mapping>OrderHeaderTransformMapping</mapping>
         </transformation>
      </transformations>
</task>
```

The above sample specifies a transformation that will transform the OrderHeader set read from InData source and create a new NewOrderHeader set in the task data.

Attributes in "transformation"

Element Name	*Description*	*Required*
`type`	Type of the transformation "**mapping**" a transformation using a mapping "**script**" a transformation using a script	No (default is "*mapping*")
`set`	Name of the set to be transformed	No
`output-set`	If the set is to be transformed into a new set, then use this attribute to define the destination set name. The transformed task data will contain a new set with this name specified. If this attribute is not specified, then the new transformed set, will replace the input set.	No
`mode`	Normally all of the transformations, *by default*, are executed just before the data is written to its destination data source. However using this attribute, the mode (or the execution sequence) of a transformation can be configured. The available modes are as follows. "**source**": The transformation is executed just after the data is read from the source. "**oldsource**" : The transformation is executed just after the data is read from the old source. The old source data is compared with the source (latest data) for difference processing. "**destination**" : The transformation is executed just after the data is written to its destination.	No (default is "*destination*")

Elements in "transformation"

Element Name	*Description*	*Required*
`mapping`	Name of the mapping to be used in transformation. This is used of the type is of a "mapping".	Yes
`script-name`	Name of the script to be executed for the transformation. This is used only if the type is of a "script".	
`record-state`	This optional element allows the change of record state of all the records in the set to be modified to the state specified by the element. The valid values are as follows.	No

	A = Added – All the records will be marked as added (new) M = Modified – All the records will be marked as modified (updated) D = Deleted – All the records will be marked as deleted (removed)	
`ignore-add`	This optional element allows the filtering the records with the "added" (new) record status to be excluded from the set (i.e. be removed from the set). The valid values are "yes" or "no".	No (Default value is "no" if not present)
`ignore-modified`	This optional element allows the filtering the records with the "modified" (updated) record status to be excluded from the set (i.e. be removed from the set). The valid values are "yes" or "no".	No (Default value is "no" if not present)
`ignore-deleted`	This optional element allows the filtering the records with the "deleted" record status to be excluded from the set (i.e. be removed from the set). The valid values are "yes" or "no".	No (Default value is "no" if not present)

8.3 Data Sets Configuration

Data sets define the locations where the actual data resides. The following is an excerpt of the data-set layout:

```
<data-set name="OrdersFile" type="File">
                <!—data-set details -->
</data-set>
```

Each data-set must have a **name** and a **type** attribute. An optional attribute is **limit**. This will activate data chunking. See section 3.1 Data Iteration and Chunking for further details.

Attributes of "data-set"

Attribute	*Description*	*Required*
`name`	The name of the data-set to be used by the task.	Yes
`type`	The type of the data-set. Can be a file, BrightServer, Pronto, JDBC dara-set, Email, Push or BrightForms accessor.	Yes
`limit`	Defines the data chunking size.	No

There are several types of data-sets that are available; these include ASCII files, BrightServer™ tables, JDBC data sources, BrightForms accessor and more. Each data-set has its own XML layout which will be detailed in the following sub-sections.

8.3.1 File

A File data-set contains a **sets** element, which itself contains one or more **set** elements. Each **set** corresponds to a file. The following example extends the orders use case, having two sets/files, comprising the new order data.

```
< data-set name="OrdersFile" type="File">
   <sets>
      <set name="OrderHeader">
         <file-name>E:/bi2/test/ORDER_HEADER.TXT</file-name>
         <mapping>OrderHeaderCSVMapping</mapping>
      </set>
      <set name="OrderItem">
         <file-name>E:/bi2/test/ORDER_ITEM.TXT</file-name>
         <mapping>OrderItemCSVMapping</mapping>
      </set>
   </sets>
</ data-set>
```

Attributes of File "set"

Attribute	***Description***	***Required***
`name`	The name of the set.	Yes

Elements in File "set"

Element Name	***Description***	***Required***
`file-name`	Contains the name of the file to be read or written to. This filename may contain wildcard characters * and ? When writing, the filename may contain the BrightIntegrator value marker, for producing variable file names. See section 8.3.6.1 for details. When reading, if the filename starts with "http" then the file will be treated as being read from a URL using http.	Yes
`mapping`	The name of the mapping that will be used to map the data fields in the file, and define the data types. See overview in section 4.0 Data Mapping, and details in section 7.4 Mappings Configuration	Yes
`append`	If set to yes, then when writing to this data-set, information will be appended, if the file already exists. Otherwise the file will be rewritten from the beginning.	No; defaults to "yes".
`include-header`	If set to 'yes', then when writing to this data-set, and if it is a CSV or Fixed file, and if the file is being created without append, then the first line will be the field names either delimited by separators or in fixed length fields. If the data-set is being read, then the first line will be ignored.	No; defaults to false/no
`after-read-delete`	If set to "yes" then every time this set/file has been successfully read, the source file is deleted. This minimises the housekeeping of files created by BrightIntegrator.	No

`after-read-copy-to`	Specifies a file name. If this element is specified, then every time this set/file has been successfully read, the source file is copied using the specified file name.	No
`after-write-copy-to`	Specifies a file name. If this element is specified, then every time this set/file is successfully written, the destination file is copied using the specified file name.	No
`ignore-not-exist`	By default, an exception is thrown when the set/file to be read does not exist. If this element is given as true/yes, and if the file is missing for this set, then this set/file will be ignored for reading data.	No; defaults to false/no
`multi-file`	Accepted values “first-match” or “all”. This element defines the behaviour when wildcard characters appear in the file-name element. First-match is the default behaviour, which means that the first file matching the filename pattern will be taken as the filename for this set. If the element value is “all”, then all files matching the filename pattern will be read, and all their data appended to the set.	No ; defaults to first-match
`xslt-file`	The mapping element must be set to “xml” for this element to apply. This element optionally specifies a file name for the XSL transform that will be applied to the XML output, before the output file is written.	No
`fop-output`	The xslt-file element must be specified for this element to apply. This element optionally enables processing of XSL Formatting Objects (XSL-FO). The result from the XSL transformation is assumed to be a Formatting Object tree, and BrightIntegrator renders the resulting pages in the specified format. For details, see Appendix E – Formatting Objects.	No, accepted values: “pdf”, “mif”, “pcl”, “ps”, “txt”, “svg”, ”print”

8.3.2 BrightServer™

A BrightServer™ data-set contains various elements which identify a server, its connection parameters, and name the query to be run on the server. It also contains a **sets** element, which itself contains one or more **set** elements. Each set corresponds to a server table. Here is an example:

```
<data-set name="ServerTable1" type="BrightServer" limit=64>
      <url>localhost:8080</url>
      <username>bsadmin</username>
      <password>changeit</password>
      <query>Query1</query>
      <sets>
            <set name="TABLE1">
                  <table-name>TABLE1</table-name>
                  <mapping>Query1</mapping>
            </set>
      </sets>
</data-set>
```

Note: BrightServer™ data-sets behave slightly different with respect to the data-set **limit** attribute. If a non-zero positive number is specified, then data will always be read in chunks of 64. So the value for the **limit** is effectively overridden to be 64.

Elements in BrightServer "data-set"

Element Name	*Description*	*Required*
`url`	The IP address and port number for the BrightServer™.	Yes.
`use-ssl`	Instructs BrightIntegrator to initiate connection to BrightServer via a secure https port specified in the ***url*** string above.	No Defaults set to "***no***"/"***false***"
`use-compression`	Instructs BrightIntegrator to turn the compression ON when communicating with BrightServer. All the data will be compressed before sent to server.	No Defaults set to "***no***"/"***false***"
`username`	The user name that will be used to log in.	Yes.
`password`	The password that will be used to log in.	Yes.
`query`	The name of the query that will be run on the BrightServer™. See Section 6.5 Queries	Yes
`is-incremental`	If set to Yes, then only the incremental data will be returned by the read. Otherwise, it will read everything from this data set.	No Defaults to "***no***"/"***false***"
`sets`	Contains one or more **set** elements.	Yes

Attributes of BrightServer "set"

Attribute	*Description*	*Required*
`name`	The name of the set that corresponds to a server table.	Yes

Elements in BrightServer “set”

Element Name	*Description*	*Required*
`table-name`	The name of the table for the set. This table will feature in the data-set query above.	Yes.
`mapping`	The name of the query mapping that will be used to map the data fields in the server table, and define the data types. See overview in section 4.0 Data Mapping, and details in section 7.4 Mappings Configuration	Yes

8.3.3 JDBC

A JDBC data-set contains various elements which identify a server, its connection parameters, and name the query to be run on the server. It also contains a **sets** element, which itself contains one or more **set** elements. Each set corresponds to a server table.

Here is an example:

```
<data-set name="JDBCBarCodeTable" type="jdbc" limit="5">
      <url>jdbc:microsoft:sqlserver://server:port;
            DatabaseName=dbname;SelectMethod=cursor</url>
      <username>user</username>
      <password>password</password>
      <jdbc-driver>com.microsoft.jdbc.sqlserver.SQLServerDriver
      </jdbc-driver>
      <query>BarcodeQuery</query>
      <sets>
            <set name="tblBarcode">
                  <table-name>tblBarcode</table-name>
                  <mapping>BarcodeQuery</mapping>
            </set>
      </sets>
</data-set>
```

Elements in JDBC "data-set"

Element Name	*Description*	*Required*
url	The JDBC URL used to connect to the data source. This defines the database connection string.	Yes.
username	The user name that will be used to log in.	Yes.
password	The password that will be used to log in.	Yes.
jdbc-driver	The name of the JDBC driver to be used to connect to this data source. BrightIntegrator™ will load and instantiate this driver, via the JDBC driver manager.	Yes.
query	The name of the query that will be run on the data source. See Section 7.5 Queries for more details.	Yes
sets	Contains one or more set element.	Yes

Attributes of JDBC "set"

Attribute	*Description*	*Required*
name	The name of the set.	Yes

Elements in JDBC "set"

Element Name	*Description*	*Required*
table-name	The name of the table for the set. This table will feature in the data-set query above.	Yes.
mapping	The name of the mapping that will be used to map the data fields in the server table, and define the data types. See overview in section 4.0 Data Mapping, and details in section 7.4 Mappings Configuration.	Yes

query	The name of the query to be used to read this set. If this element is not specified, then the main query of the data source is used. If this is specified, then the main query of the data source is not used (Since Version 3.1.0)	Optional

8.3.3.1 Difference between the main <query> of the data-set and <query> element of sets ?

When reading a single table using the JDBC accessor there is no difference between those two elements. If the <query> element is not defined for the set, then BrightIntegrator will use the <query> element defined for the <data-set> element. If the <set> element has its own <query> element then, the main <query> will not be used at all.

There is however a very important difference when reading multiple tables (sets) from the JDBC accessor. The <query> element defined for each <set> element will be used to define the SQL query to fetch the result set from the JDBC data source. If a <query> element is defined for the set, each set (i.e. each table), will use its own <query> element and <mapping> to fetch the data, and the main <query> will be ignored.

If multiple tables are read, and each <set> does not have a <query> element defined for it, then the main <query> element will be used repeatedly for each set to be read from the JDBC source. The way the <query> is used in the instance is as follows.

Let's assume we have a parent table (table-1) and several children (table-2 ... table-n). The following pseudo code is used to generate the SQL SELECT statement to fetch the data.

```
For each table-x
    Include table-i mapping output as defined  by the <mapping> element
        For i=0 to i-x
            Append conditions defined for table-i
            If (i>0) append table relationship for table-(i-1) → table-i
```

Example for a query containing P (parent table), C1 (child 1) and C2 (child 2) is given below.

To fetch P records the following SELECT statement will be constructed.

```
    Select <output columns defined for P>
    From P
    Where <P column conditions>
```

To fetch C1 records the following SELECT statement will be constructed.

```
    Select <output columns defined for C1>
    From P, C1
    Where <P column conditions> AND
        <P→C1 relationship> AND <C1 column conditions>
```

To fetch C2 records the following SELECT statement will be constructed.

```
Select <output columns defined for C1>
From P, C1, C2
Where <P column conditions> AND
        <P→C1 relationship> AND <C1 column conditions> AND
        <C1→C2 relationship> AND <C2 column conditions>
```

IMPORTANT NOTE: When defining query conditions, the above SQL generation must be kept in mind. Since certain conditions may be included or excluded depending upon which table is being read, some illegal SQL statements may be generated by BrightIntegrator as a consequence.

8.3.3.2 How to specify parameterized date-time ranges for JDBC queries?

In order to specify a parameterised value which is determined at the run time, $BV$ Bright Software value markets can be used in the value string by the ***<value>*** element in the query <condition> section. The possible combinations are as follows.

$BV$_DATE_; *x*$BV$
Where ***x*** is the minutes to be added (if a positive number is specified), or to be subtracted (if a negative number is specified) from the current system time.

Example : For instance last 24 hours period can be specified using (60*24 = 1440 minutes, a negative value specifies a date time value which is 1440 earlier than the current time)

$BV$_DATE_; -1440$BV$

$BV$_DATE_MIDNIGHT_; *x*$BV$
Where x is the minutes to be added (if a positive number is specified), or to be subtracted (if a negative number is specified) from the last midnight time.

Example : For instance 12pm today can be specified using the following (60 minutes*12 hours from midnight= 720 minutes, a positive value specifies a 12 hour addition to the midnight)

$BV$_DATE_MIDNIGHT_; 720$BV$

IMPORTANT NOTE: These $BV$ values are only available for the JDBC queries.

8.3.4 Pronto

Bright Integrator can also connect to a Pronto data set via the PIE Connector. This allows multiple API calls to the PIE, thus it has an additional configuration file for the API methods to be called. Read Appendix A for further information on the API configuration file.

The Pronto data-set contains elements to identify the Pronto server, the location of the Pronto API config file, the main set mapping to be used and the temporary file location details. Here is an example:

```
<data-set name="ProntoServer" type="Pronto" >
      <url>ProntoServerIP:1977</url>
      <config-file>c:/bi2/conf/pronto_config.xml</config-file>
      <main-set-mapping>OrderHeaderCSVMapping</main-set-mapping>
      <tx-file-location>c:\bi2\conf\</tx-file-location>
      <sets>
            <set name="Number">
                  <mapping>ApiMapping</mapping>
            </set>
      </sets>
</data-set>
```

Elements in Pronto “data-set”

Element Name	***Description***	***Required***
url	The IP address and port number for the PIE server.	Yes.
config-file	The location of the additional config file dedicated for this data-set. This file basically defines which Pronto API methods are to be called, in response to the incoming data. See Appendix A: API Configuration File for details.	Yes.
main-set-mapping	The name of the set that at the top of the relationship tree. This is used for when grouped data is being consumed.	Yes.
tx-file-location	The directory that will be used to store some temporary files. Some files are written to store state information regarding what data has been successfully consumed. This is useful in the case where an error occurs during part of the way processing.	Yes.
sets	Contains one or more set element.	No
submit-empty-data	By default empty data is not submitted, hence pre and post task APIs are not executed. Use this element to override the behaviour. When set to “yes”, BrightIntegrator will execute the pre and post task APIs even though the task data to be written is empty, i.e. it does not contain any data for writing.	No Default set to “no”

Attributes of Pronto “set”

Attribute	***Description***	***Required***

name	The name of the set.	Yes

Elements in Pronto "set"

Element Name	***Description***	***Required***
table-name	The name of the table for the set. This table will feature in the data-set query above.	Yes.
mapping	The name of the mapping that will be used to map the data fields in the server table, and define the data types. See overview in section 4.0 Data Mapping, and details in section 7.4 Mappings Configuration.	Yes

8.3.5 Web Services

Bright Integrator can also connect to a Web Service provider. Similarly to the Pronto data-set, it may use one or more API calls to the web service, and thus it also has an additional configuration file for the API methods to be called. Read Appendix A for further information on the API configuration file.

The Web Services data-set contains elements in common with the Pronto data-set, and further in addition, it contains elements only associated with web service details. Here is an example:

```
<data-set name="WebServer" type="webservices" >
      <url>www.dataaccess.com/webservicesserver/conversions.wso</url>
      <config-file>c:/bi2/conf/pronto_config.xml</config-file>
      <main-set-mapping>OrderHeaderCSVMapping</main-set-mapping>
      <tx-file-location>c:\bi2\conf\</tx-file-location>

<namespace-uri>http://www.dataaccess.com/webservicesserver/</namespace-uri>
      <namespace-prefix>ns1</namespace-prefix>
      <msg-type>rpc</msg-type>
      <string-return-type>yes</string-return-type>

      <sets>
            <set name="Number">
                  <mapping>ApiMapping</mapping>
            </set>
      </sets>
</data-set>
```

Elements in WebServices “data-set”

Element Name	*Description*	*Required*
url	The url for the web service. It must contain the web service context.	Yes.
config-file	The location of the additional API config file dedicated for this data-set. See Appendix A: API Configuration File for details.	Yes.
main-set-mapping	The name of the set that at the top of the relationship tree. This is used for when grouped data is being consumed.	Yes.
tx-file-location	The directory that will be used to store some temporary files. Some files are written to store state information regarding what data has been successfully consumed. This is useful in the case where an error occurs during part of the way processing.	Yes.
service-name	The SOAP action URI for Axis RPC call.	No
user-name	The user name to use for this API call.	No
user-password	The password to use for this API call.	No
namespace-uri	The URI for the name space to use for this API call.	No
namespace-prefix	The prefix that is bound to the name space.	No

use-ssl	Whether to use SSL or not for connecting to the server. Defaults to no.	No
truststore-filename	The filename for the the trust store. Used for server authentication.	No
truststore-password	The password for the trust store. Used for server authentication.	No
result-param	Names the output parameter to be interpreted as the result of the API call. Used for SOAP message web services.	No
msg-type	Either msg (SOAP message web service) or rpc (RPC type web service, use Axis Service Call). Defaults to rpc.	No
string-return-type	Flag that defines whether to interpret the return value from the RPC call as a string. Only used for RPC. Defaults to no.	No
sets	Contains one or more set element.	No
submit-empty-data	By default empty data is not submitted, hence pre and post task APIs are not executed. Use this element to override the behaviour. When set to "yes", BrightIntegrator will execute the pre and post task APIs even though the task data to be written is empty, i.e. it does not contain any data for writing.	No Default set to "no"

Attributes of Web Services "set"

Attribute	***Description***	***Required***
name	The name of the set.	Yes

Elements in Web Services "set"

Element Name	***Description***	***Required***
table-name	The name of the table for the set. This table will feature in the data-set query above.	Yes.
mapping	The name of the mapping that will be used to map the data fields in the server table, and define the data types. See overview in section 4.0 Data Mapping, and details in section 7.4 Mappings Configuration.	Yes

8.3.6 Email

BrightIntegrator can also connect to an Email data-set (i.e. send data as an email to specified recipients or read email messages from specified in-boxes). This allows BrightIntegrator to send escalation messages to the system administrator via email etc. With this email dataset, you can configure the email format and also send the specific task data.

When the data is processed by the email accessor, it can be configured the process the data either record by record (i.e. the ***record*** mode where each record is processed and sent as an email), or as a whole task data (i.e. the ***taskdata*** mode where all the data will be processed and sent in a single email). This is specified by the **<*process-by*>** element. See below.

Note that in the "record" mode, only a single set can be processed by the email accessor. If a task data with more than one set is sent to the email accessor, then the email accessor will report an exception. By default, the email accessor processes the data in the "taskdata" mode.

If the data is *grouped*, then each *group*'s data is sent as separate emails for each group.

Here is a sample Email data-set configuration:

```
<data-set name="EscalationEmail" type="Email" >
 <host>205.214.83.216</host>
 <port>25</port>
 <to>admin@mail.com.au</to>
 <from>bi@mail.com.au</from>
 <cc-list>
  <cc>it_manager@mail.com.au</cc>
 </cc-list>

 <subject>RE:Successfully sent Order OID=$BV$OID$BV$ QTY=$BV$Qty;###.##$BV$
 </subject>

 <body>
  <section execute="once" type="text">Dear customer,</section>
  <section execute="once" type="text"> </section>
  <section execute="once"
    type="text">Orders($BV$_DATE_;dd/MM/yy$BV$)</section>

  <section execute="once"
    type="text">----------------------------</section>
  <section set="OrderItem" execute="always"
    type="text">Prd=$BV$Prod$BV$ OID=$BV$OID$BV$
    QTY=$BV$Qty;###.##$BV$</section>

  <section execute="once" type="file">/bi3/data/body_section1.txt</section>
 </body>
 <attachments/>
</data-set>
```

The email body will look like this:

Dear customer,	Section 1
	Section 2
OrderItems (24/Nov/2005)	Section 3
------------------------------	Section 4
Prod==2 OID=100 QTY=1 Prod==2 OID=100 QTY=1	Section 5
Kind regards, Bright Software	Section 6

The following Email data-set is used to read the specified email account in-boxes:

```
<data-set name="Email" type="Email" >
      <host>205.214.83.216</host>
       <port>25</port>
       <sets>
            <set name="OrderItem">
                  <mapping>EmailMapping</mapping>
                  <email-accounts>
                        <email-account name="fred@slaterockandgravel.com">
                            <password>flintstone</password>
                        </email-account>
                       <email-account name="barney@slaterockandgravel.com">
                            <password>rubble</password>
                       </email-account>
                  </email-accounts>
                  <delete-server-msgs>yes</delete-server-msgs>
            </set>
       </sets>
</data-set>
```

8.3.6.1 $BV$ Value Place Holder

$BV$ is the BrightIntegrator value marker, this is used by the Data Textualizer to define and format the task data sent to the email accessor. It uses the standard number and dateTime formatting concepts defined in this document. You can use different files to define the different body section elements of the email accessor.

$BV$ places holder can be used in couple of ways.

a) A field places holder. In this case, the field name is placed between $BV$ pairs. For example if the value of the "Prod" field from the "OrderItem" set is to be textualized or printed, then $BV$Prod$BV$ is used.
b) For system wide values such as current system date and time is to be specified, then a system constants are used. Currently the following constants are defined.

Constant	Description
DATE	Returns the current system date and time
NUN	Returns the next unique number. The next unique number is calculated using the milliseconds passed since midnight 1 January 1970. BrightIntegrator ensures that a distinct number is returned each time, if a record happens to have been processed in the same millisecond.
NULL	Returns a “null” value
_DATE_MIDNIGHT_	Returns the starting time of today. i.e. today’s date with 00:00:00.000 time.

An optional format field can be specified by a $BV$ place holder. To specify the format of the field, use a semicolon (‘;’) after the field name or system constant and specify the format for the field. This format field would be for date-time value or numeric field. For example below $BV$ value will read the system time and convert it to a date-time string in dd/MM/yy format.

```
$BV$_DATE_;dd/MM/yy$BV$
```

8.3.6.2 <body> Element

<body> element is consisted of ***section***s. Each section is defined by a <section> element. The attributes of the <section> element are as follows.

Attribute	***Description***	***Required***
execute	Section’s execution type ***once*** : The section appears only once in the text ***always*** : The section is repeated for each record in the set specified by the ***set*** attribute.	Yes
type	The type specifies the source of the text. ***text*** : The section element contains the actual text. This text may contain $BV$ place holders. ***file*** : The section text is read from the file and processed. The section element contains the file name in this case.	Yes
set	Name of the set to be processed by this section. If a set name is specified, each record is converted into text using this section definition.	No

Elements in Email “data-set”

Element Name	***Description***	***Required***
host	Email host address. This can be overwritten by the	Yes
port	Email server port number. This is set to default Port “25”. Does not need to be specified.	Yes

to	Email address of the person to which the email is to be sent. This text may contain BV place holders.	Yes
from	Email address of the person who is sending the email.	Yes
cc-list	List of email addresses of the persons to which the email is to be carbon-copied to.	No
subject	Subject of the email	Yes
body	Body of the email. This is segmented into body sections. You can list as many sections as required.	No
attachments	Name of the file that can be sent with the email. This is optional.	No
process-by	This specifies how the task data is to be processed by the email accessor. Available options for this element are as follows. **record**: Every record in a recordset is sent as a email separate email **taskdata**: Whole task data is to be sent as a single email. If the data is grouped, then each group's task data is sent as a single email.	No Default = **taskdata** i.e. whole task data is sent as a single email.
sets	Contains one or more set element.	No

Elements in "cc-list"

Element Name	*Description*	*Required*
cc	Email address of the person to which the email is to be carbon-copied to. This text may contain BV place holders.	Yes

Elements in "body"

Element Name	*Description*	*Required*
section	Specifies a section of the body element that is to be executed in order of appearance.	Yes

Attributes of "section"

Element Name	*Description*	*Required*
execute	Defines if the body section is to be executed "once" or "always".	Yes
type	Defines the type of the body section, possible values are "text" or "file". If "file", the body section value should specify a file name for the body section to be executed.	Yes

Elements in "attachments"

Element Name	*Description*	*Required*

attachment	Specifies the file name to be attached to this email. This text may contain BV place holders. A physical file content or a value of the record can be attached as an attachment to the email. This is determined by the ***field*** attribute of the attachment element. See below.	Yes

Attributes of "attachment"

Element Name	*Description*	*Required*
name	The name of the attachment as it appears in the email, which can be different to the file name specified by the attachment element. If this attribute is not specified, then, by default, the file name will be used as the attachment name. This text may contain BV place holders.	No
field	Name of the record field to be sent as an attachment to the email. If this attribute is not present or empty, then the attachment element contains the name of the physical file to be sent as an attachment with the email.	No

Attributes of Email "set"

Attribute	*Description*	*Required*
name	The name of the set.	Yes

Elements in Email "set"

Element Name	*Description*	*Required*
email-accounts	Contains one or more email-account elements.	Yes.
mapping	The name of the mapping that will be used to map the email messages.	Yes

Attributes of Email "email-account"

Attribute	*Description*	*Required*
name	The name of the email account, which is used to log in to the mail server and access the user inbox.	Yes

Elements in Email "email-account"

Element Name	*Description*	*Required*
password	The password of the email account, which is used to log in to the mail server and access the user inbox.	Yes.

8.3.7 Push

Bright Integrator can also connect to a Push data-set. This allows BrightIntegrator to use the Push Module and send published data to a list of subscribers. The Push data-set simply defines the configuration file to be used by the Push Module.

Here is an example:

```
<data-set name="Push" type="Push" >
      <config-file>/bi/conf/fileconfig.xml</config-file>
</data-set>
```

Elements in Push “data-set”

Element Name	*Description*	*Required*
config-file	Name of the Push Module configuration file to be executed by the P&S module.	Yes

See Section 5 and 8 for details on the Push Module elements and attributes.

8.3.8 BrightForms

Bright Integrator can also connect to a BrightForms data set. This allows the job processor to send and receive data to and from a BrightForms client.

The following is a sample XML configuration for a BrightForms data-set.

```
<data-set name="PushToBF" type="brightforms">
<server>192.168.39.58</server>
<port>8080</port>
<message-type>data</message-type>
<use-compression>yes</use-compression>
<client-column-names>FIELD1,FIELD3</client-column-names>
<client-column-values>_#BFValue_UN_,LocalDesc</client-column-values>
</data-set>
```

Elements in BrightForms "data-set"

Element Name	*Description*	*Required*
server	The IP address of the BrightForms client.	Yes
port	The port number on which the client listens for the messages.	Yes
message-type	Specifies whether the data will be pushed with the message or the name of the sync rule to be sent to to the BrightForms client. Possible types are "**data**" or "**sync-rule**"	Yes
client-column-names	This attribute is used if the message-type is "**data**". It specifies the names of the client columns of which values are passed from the BrightForms data-set to the client. This is used to populate the required client columns that does not exists in the actual task data. See the table below for the definition of BrightForms Special System Column Values.	No
client-column-values	These are the actual client column values used by the BrightForms client to populate the local client columns in the database. This attribute is only used if the message-type is "**data**". *Please see the table below for generating special values using the special system column values.*	No
sync-rules	This lists the names of the sync rules that will be used by the BrightForms client to pull data from the server. It may contain multiple comma separated sync rule names. This attribute is used only if the message-type attribute is "**sync-rule**".	Yes if message-type is "sync-rule"
use-compression	Specifies if the data is to be compressed if the message-type is "**data**".	No

BrightForms Special System Column Values
BrightForms will generate the corresponding values, when it encounters these special place holders

Value Holder	Description
_#BFValue_UN_	Generates and uses the next unique number for the column value
_#BFValue_Null_	Puts "***null***" as the column value
_#BFValue_SD_	Puts the current **S**ystem **D**ate and time as the column value

NOTE: Column names are case *insensitive*.

<bf-message> format

BrightForms accessor will operate in "writer" mode and push the data to the designated BrightForms client. It will convert the task data to the standard BrightForms packed XML format and send it using the <bf-message>.

Using the <bf-message>, BrightForms accessor can push actual data or get BrightForms to pull data by specifying the sync rules that BrightForms needs to execute. The message format for pushing task data is given below. Note that ***type*** attribute is set to "*data*". The message contains a **task-data** element which will contain the standard BrightForms data element for sending records to BrightForms. If the ***compression*** attribute is set to "on", then it will contain the compressed data element in base64 format.

```
<bf-message version="1.0" type="data">
    <task-data compression="off">
      <data>
       <table name="TABLE1">
        <columns>
         <col type="int">FIELD1</col>
         <col type="string">FIELD2</col>
         <col type="string">FIELD3</col>
        </columns>
        <records>
         <record>
           <item>10</item>
           <item>Item 10</item>
           <item>Description 10</item>
         </record>
         <record>
           <item>11</item>
           <item>Item 11</item>
           <item>Description 11</item>
         </record>
         <record>
           <item>12</item>
           <item>Item 12</item>
           <item>Description 12</item>
         </record>
        </records>
       </table>
      </data>
    </task-data>
</bf-message>
```

For triggering data pull by BrightForms an another type <bf-message> is available, which is also given below. This message type is configured by setting the ***type*** attribute to "*sync-rule*".

The ***sync-rule*** element within the message contains the comma separated name of the sync rules to be executed by the BrightForms to pull data.

```
<bf-message version="1.0" type="sync-rule">
   <sync-rule>SyncGetJobs,SyncGetCustomerList </sync-rule>
</bf-message>
```

Background synchronisation and BrightIntegrator

The use of BrightForms accessor in BrightIntegrator should also be incorporated within the mobile application created in BrightBuilder. If using data as the message-type, then the Push Listener option in the mobile application should be enabled. If using sync-rule as the message-type, then the Background Synchroniser option should also be enabled.

With the Push Listener and Background Synchroniser enabled, this allows the BrightForms client to listen for the sync-rule to be executed and run a background synchronisation.

The sync rules used in the BrightIntegrator configuration file should have already been enabled and made as a background sync rule in the application. When BrightIntegrator triggers the data pull to BrightForms, BrightForms will then trigger the background synchronisation process to send the data to BrightServer.

There are several considerations to be made if using the data pull mechanism of BrightIntegrator, these are as follows:

- Both the "Enable Background Synchroniser" and "Enable Push Listener" properties of the application project is true.
- All sync rules to be executed has been enabled and the background sync-rule flag is true.
- All the devices have unique IP addresses.

Read more about the Background Synchronisation process from BrightBuilder's Users Manual.

8.3.9 Script

Using JavaScript in the system will provide extreme flexibility and power. This will enable users to consume or provide data to and from non-conventional data sources.

The scripting support will be provided by using the standard existing “accessor” architecture. This fits seamlessly into existing BrightIntegrator world. Using this new accessor will be used in the same manner similar to a JDBC accessor. It will be instantiated and used by the sync engine as normal.

The following is a sample XML configuration for a script data-set.

```
<data-set name="MyScriptDataSource" type="script">
     <script-name>myscript</script-name>
     <query-name>MyScriptQuery</query-name>
     <sets>
             <set name="BI_TEST">
                 <mapping>Mapping_BI_TEST</mapping>
             </set>
             <set name="BI_TEST_CHILD">
                 <mapping>Mapping_BI_TEST_CHILD</mapping>
             </set>
    </sets>
</data-set>
```

Elements in BrightForms “data-set”

Element Name	*Description*	*Required*
script-name	Name of the script to be executed	Yes
query-name	Name of the query that will provide payload. The query will most likely a user defined one. See section 8.5.1	No
sets	Contains one or more set elements. If they are not defined, then the script is expected to return mapping (FieldInfo) back to accessor.	No

Attributes of script “set”

Attribute	*Description*	*Required*
name	The name of the set.	Yes

Elements in script “set”

Element Name	*Description*	*Required*
mapping	The name of the mapping that will be used to map the data fields. This could any of the following mapping types: Query, CSV file, or fixed file.	Yes

8.4 Mappings Configuration

The **mappings** element contains one or more **mapping** elements. The purpose of a mapping is to provide a unique, logical name for each data field. A **mapping** element must contain a **name** and a **type** attribute. It also contains a **fields** element, containing **field** elements.

Each **field** element must contain a **name** and (data) **type** attribute. Optionally a field can be declared as being a primary key (**pk**=1).

The possible internal data types for each field are: (the names themselves are case-insensitive)

- string
- int
- boolean
- double
- dateTime
- base64Binary
- rawBinary

There are several types of mappings. They will be detailed in the following sub-sections.

8.4.1 CSV (Character Separated Value) File Mapping

The CSV file type mapping provides the details about the field specific format, as well as the data type mapping. An example of a CSV file type mapping follows:

```
<mapping name="BarCodeCSVMapping" type="csv">
      <fields sep="," esc='"'>
            <field name="barcode" type="string" pk='1'>
                  <format/>
            </field>
            <field name="stock_code" type="int" pk='1'>
                  <format>00000000</format>
            </field>
            <field name="description" type="string">
                  <format/>
            </field>
      </fields>
</mapping>
```

Attributes of CSV "fields"

Attribute	*Description*	*Required*
sep	Defines the character that will be used to separate the values in the file.	No; defaults to comma
esc	Defines the character used to escape from the separator character. This is useful if the data itself contains the separator character.	No; defaults to double quote.

`always-esc`	If this flag is set to "*yes*", then all the fields in the CSV output will be enclosed with the escape character specified by the "***esc***" attribute (See above).	No; defaults always to "*no*"

The "esc" character acts as a delimiter when the separator or escape character is embedded within the field values. The following rules are applied when inserting the escape character:

1. Each field may be enclosed in double quotes if you wish, i.e. write a field as `the dog` or `"the dog"`.
2. Any field that contains a comma **must** be surrounded by quotes.
3. Any field that contains double quotes (") **must** be enclosed in double quotes and escape the double quotes in the field by preceding it with another double quote, e.g. the field `big "brown" fox` should be entered as `"big ""brown"" dog".`
4. Spaces within a field are significant, i.e. the 2 fields `the piano` and `the  piano` are not equivalent since the second one contains two spaces between each word.
5. If a field is quoted, there **cannot** be any spaces between the leading and trailing commas and the enclosing double quotes, i.e. the two consecutive fields the dog and the cat should be entered as `"the dog","the cat"` or `the dog,the cat` and not `"the dog" , "the cat".`

For example, a table contains the following values:

ID	NAME	NOTES
1	Jane	I have a cat, a dog and a bird.
2	John	He sang "Moon River"
3	Grey	No notes

If these records were exported using a csv mapping with comma as the separator and double-quotes (") as the escape character, the file will look like this:

```
1,Jane,"I have a cat, a dog and a bird."
2,John,"He sang """Moon River"""
3,Grey,No Notes
```

If the "always-esc" is set then the output would like as follows.

```
"1","Jane","I have a cat, a dog and a bird."
"2","John","He sang """Moon River"""
"3","Grey","No Notes"
```

Attributes of CSV "field"

Attribute	***Description***	***Required***
`name`	The name of the field, which will be used to identify the data elsewhere in the configuration file.	Yes
`type`	The internal data type of the field.	Yes
`status`	Set to true indicates that this field holds the change status for the whole record. The value	No; defaults to false/no.

	may be "A" for added, "M" for modified, or "D" for deleted. In this way, a diff result may be written to file, or an incremental file may be read.	
`pk`	Set to true/yes, defines this field as a primary key.	No; defaults to false/no.
`file-external`	Set to true/yes, defines this field as sourcing its data in an external file.	No; defaults to false/no.
`file-path`	If file-external is true, then this attribute defines the path containing the external files for this field.	No; enabled by file-external.
`file-name-embedded`	If file-external is true, then this attribute set to true/yes declares that the file-name to be used for this field is embedded in the main file.	No; enabled by file-external. Default is No.
`file-path-embedded`	Used only for writing. If file-external is true, then this attribute set to true/yes declares that the file-path to be used for this field is embedded in the main file being written.	No; enabled by file-external. Default is No.
`file-name`	If file-external is true, then this attribute contains the filename for this field with file-path. This filename may use the BrightIntegrator value marker to create variable file names for each record.	No; enabled by file-external. Can be omitted if file-name-embedded is set to true/yes
`file-must-exist`	If file-external is true, and this attribute is set to true/yes, then if the external file for this field does not exist, a error will be generated. Used for only for reading.	No; enabled by file-external. Default is No.

Elements in CSV "field"

Element Name	***Description***	***Required***
`format`	The format of the field. See section 8.9 Data Value Formatting	Yes.
`trim`	Specifies if the string field value will be trimmed.	No
`convert-to-null`	If this element is present, then its string value is used in this field to interpret null values.	No
`compress`	This is an instruction to the file accessor to compress the field value when reading or writing. If this element is present and set to "yes", then when reading a binary file, the content of the file will be read and compressed before assigned to the field; when writing the content of the binary field it will be compressed before it is written to the output file.	No
`decompress`	This is an instruction to the file accessor to compress the field value when reading or writing.	No

	If this element is present and set to “yes”, then when reading a binary file, the content of the file will be read and decompressed before assigned to the field; when writing the content of the binary field it will be decompressed before it is written to the output file.	

8.4.2 Fixed-field-length File Mapping

The fixed-field-length file type mapping provides the details about the field specific format, as well as the data type mapping.

An example of a fixed-field-length file type mapping follows:

```
<mapping name=" BarCodeFixedMapping" type="fixed">
      <fields>
            <field name="barcode" type="string" pk='1'>
                  <format/>
                  <start>1</start>
                  <length>24</length>
                  <pad-char/>
            </field>
            <field name="stock_code" type="string" pk='1'>
                  <format/>
                  <start>25</start>
                  <length>10</length>
                  <pad-char/>
            </field>
            <field name="description" type="string">
                  <format/>
                  <start>36</start>
                  <length>1</length>
                  <pad-char/>
            </field>
      </fields>
</mapping>
```

Note: The **length** of the last field in a fixed-field-length file is effectively ignored. When the last field is to be read, the data reader will read all of the remaining characters on the line, for parsing the field value.

Attributes of fixed-field length “field”

Attribute	*Description*	*Required*
`name`	The name of the field, which will be used to identify the data elsewhere in the configuration file.	Yes
`type`	The internal data type of the field.	Yes
`status`	Set to true indicates that this field holds the change status for the whole record. The value may be “A” for added, “M” for modified, or “D” for deleted. In this way, a diff result may be written to file, or an incremental file may be read.	No; defaults to false/no.
`pk`	Set to true/yes, defines this field as a primary key.	No; defaults to false/no.

`file-external`	Set to true/yes, defines this field as sourcing its data in an external file.	No; defaults to false/no.
`file-path`	If file-external is true, then this attribute defines the path containing the external files for this field.	No; enabled by file-external.
`file-name-embedded`	If file-external is true, then this attribute set to true/yes declares that the file-name to be used for this field is embedded in the main file.	No; enabled by file-external. Default is No.
`file-path-embedded`	Used only for writing. If file-external is true, then this attribute set to true/yes declares that the file-path to be used for this field is embedded in the main file being written.	No; enabled by file-external. Default is No.
`file-name`	If file-external is true, then this attribute contains the filename for this field with file-path. This filename may use the BrightIntegrator value marker to create variable file names for each record.	No; enabled by file-external. Can be omitted if file-name-embedded is set to true/yes
`file-must-exist`	If file-external is true, and this attribute is set to true/yes, then if the external file for this field does not exist, a error will be generated. Used for only for reading.	No; enabled by file-external. Default is No.

Elements in fixed-field length "field"

Element Name	*Description*	*Required*
`format`	The format of the field. See section 8.9 Data Value Formatting	Yes.
`start`	The (one-based) index of the first character.	Yes.
`length`	The character length of the field.	Yes.
`pad-char`	The character that is used to pad the field, should the data content be shorter then the length of the field.	Yes.
`trim`	Specifies if the string field value will be trimmed.	No
`convert-to-null`	If this element is present, then its string value is used in this field to interpret null values.	No.
`compress`	This is an instruction to the file accessor to compress the field value when reading or writing. If this element is present and set to "yes", then when reading a binary file, the content of the file will be read and compressed before assigned to the field; when writing the content of the binary field it will be compressed before it is written to the output file.	No
`decompress`	This is an instruction to the file accessor to decompress the field value when reading or writing.	No

	If this element is present and set to “yes”, then when reading a binary file, the content of the file will be read and decompressed before assigned to the field; when writing the content of the binary field it will be decompressed before it is written to the output file.	

8.4.3 XML File Mapping

The XML file type mapping specifies the as well as the data type mapping. This mapping simply instructs for the data to be formatted in XML.

There are no definable fields in this mapping, since the data is self described in the XML, as meta-data.

The XML output conforms to the TaskData XML format. Refer to Appendix F for details.

In combination with this mapping type, the file data set can optionally define an XSL transform file that will be applied before the output file is written.

```
<mapping name="XMLMapping" type="xml"/>
```

8.4.4 Query Mapping

The query mapping maps from the server table column names and data types to the internal field names and data types. An example of a query type mapping follows:

```
<mapping name="JDBCQuery" type="query">
      <fields>
      <field name="F_INT" type='int' pk='1'>
            <column-name>F_INT</column-name>
      </field>
      <field name="F_STRING" type='string'>
            <column-name>F_STRING</column-name>
      </field>
      <field name="F_BOOLEAN" type='boolean'>
            <column-name>F_BOOLEAN</column-name>
      </field>
      <field name="F_DOUBLE" type='double'>
            <column-name>F_DOUBLE</column-name>
      </field>
      <field name="F_DATETIME" type='datetime'>
            <column-name>F_DATETIME</column-name>
      </field>
      </fields>
</mapping>
```

Attributes of “field”

Attribute	*Description*	*Required*
`name`	The name of the field, which will be used to identify the data elsewhere in the configuration file.	Yes
`type`	The internal data type of the field.	Yes
`pk`	Set to true/yes, defines this field as a primary key.	No; defaults to false/no.

Elements in “field”

Element Name	*Description*	*Required*
`table-name`	Name of the table from which the columns is sourced (Since version 3.1.0).	Optional
`column-name`	The name of the table column for the field.	Yes.
`compressed`	Option element that tells BI if the field contains compressed data. If this field is being read, then the data will be uncompressed after it is read. If the field is being written, then the data will be compressed before it is written,	No.
`compress`	This is an instruction to the JDBC accessor to compress the field value when reading or writing.	No

	If this element is present and set to “yes”, then when reading a binary file, the content of the column will be read and compressed before assigned to the field; when writing the content of the binary field it will be compressed before it is written to the destination column.	
`decompress`	This is an instruction to the JDBC accessor to compress the field value when reading or writing. If this element is present and set to “yes”, then when reading a binary blob column, the content of the column will be read and decompressed before assigned to the field; when writing the content of the binary field it will be decompressed before it is written to the destination column. This is equivalent to “compressed” option. They can be used interchangeably to read a compressed column value and decompress it.	No

8.4.5 API Mapping

The API mapping maps from API call parameters and data types to the internal field names and data types. An example of an API type mapping follows:

```
<mapping name="ApiMapping" type="api">
      <fields>
      <field name="F_INT" type='int' pk='1'>
            <param-name>paramInt</param-name>
      </field>
      <field name="F_STRING" type='string'>
            <param-name>paramString</param-name>
      </field>
      <field name="F_BOOLEAN" type='boolean'>
            <param-name>paramBoolean</param-name>
      </field>
      <field name="F_DOUBLE" type='double'>
            <param-name>doubleParam</param-name>
      </field>
      <field name="F_DATETIME" type='datetime'>
            <param-name>datetimeParam</param-name>
      </field>
      </fields>
</mapping>
```

Attributes of "field"

Attribute	***Description***	***Required***
`name`	The name of the field, which will be used to identify the data elsewhere in the configuration file.	Yes
`type`	The internal data type of the field.	Yes
`pk`	Set to true/yes, defines this field as a primary key.	No; defaults to false/no.

Elements in "field"

Element Name	***Description***	***Required***
`param-name`	The name of the table column for the field.	Yes.
`param-type`	Optional element, accepted value is "set". In this case the incoming data from the web service is interpreted as an XML string for this output parameter. The XML will be expected to be a TaskData XML object. Otherwise, an XSL transform can be applied beforehand, see below.	No.
`xslt-file`	Optional element, param-type must be set. If this element is included then it names the file that contains the XSL transform that will be applied to the XML data coming back from the web service. The result will be interpreted as a TaskData XML object.	No.

8.4.6 Email Mapping

The Email mapping maps from email messages to the internal field names and data types. An example of an Email type mapping follows:

```
<mapping name="EmailMapping" type="email">
      <fields>
            <field name = "ID"          type='string' pk='1'>
                  <source>msg-id</source>
                  <length>50</length>
            </field>
            <field name = "To"          type='string' pk='1'>
                  <source>to</source>
                  <length>200</length>
            </field>
            <field name = "From"        type='string'>
                  <source>from</source>
                  <length>200</length>
            </field>
            <field name = "DateSent"    type='datetime'>
                <source>date</source>
            </field>
            <field name = "Subject"     type='string'>
                  <source>subject</source>
                  <length>100</length>
            </field>
            <field name = "Body"        type='string'>
                  <source>body</source>
                  <length>4096</length>
            </field>
            <field name = "Attach"      type='string'>
                  <source>attachment-list</source>
                  <length>4096</length>
            </field>
      </fields>
</mapping>
```

Attributes of “field”

Attribute	***Description***	***Required***
`name`	The name of the field, which will be used to identify the data elsewhere in the configuration file.	Yes
`type`	The internal data type of the field.	Yes
`pk`	Set to true/yes, defines this field as a primary key.	No; defaults to false/no.

Elements in “field”

Element Name	*Description*	*Required*
`source`	The message item to be used for this field. Acceptable entries are: • msg-id – unique identifier for the message • to – recipient of the email message • from – sender of the email message • date – date/time that email was • subject – of the email message • body – of the email message • attachment-list – a comma separated list of the names of the email attachments. Only the names are passed, not the attachment data items themselves.	Yes.
`length`	Optional element. If this is specified, then it will limit the length of the data that will be passed on. For example, if the length of the email message body is specified as “4096”, then the maximum length string that be passed in the body field will be 4096 characters.	No.

8.4.7 Text File Mapping

The file mapping is used to convert task data to a text. An example of a query type mapping follows:

```
<mapping name="OrderHeaderTextMapping" type="text">
  <body>
    <section execute="once" type="text">Dear $BV$Cust$BV$,</section>
    <section execute="once" type="text"> </section>
    <section execute="once" type="text">-----------------------</section>
    <section execute="always" type="text">Order==$BV$OID$BV$</section>
    <section execute="once" 
type="file">/cvs/bi2/testdata/body_section1.txt</section>
   </body>
</mapping>
```

This mapping uses the data to textualizer feature of BrightIntegrator that is also used by the email accessor. Please see Section 8.3.5 for the definition of the <body> element.

8.4.8 Transformation Mapping

This the mapping definition used to transform task data sets:

```
<mapping name="HeaderTransformMapping" type="transformation">
      <fields>
            <field name = "OID" type='int' pk='1'>
                  <src-type>bi</src-type>
                  <set-name>OrderHeader</set-name>
            </field>
            <field name = "Cust" type='string'>
                  <src-type>bi</src-type>
                  <set-name>OrderHeader</set-name>
            </field>
            <field name = "Extra" type='string'>
                  <src-type>bi</src-type>
                  <set-name>OrderHeader</set-name>
            </field>
            <field name = "CustDuplicate" type='string'>
                  <src-type>bi</src-type>
                  <set-name>OrderHeader</set-name>
                  <field-name>Cust</field-name>
                  <post-function>ReplaceChar,176,"DegreeC"</post-function>
            </field>
            <field name = "NewString" type='string'>
                  <src-type>constant</src-type>
                  <value>AU</value>
            </field>
            <field name = "NewDate" type='datetime'>
                  <src-type>constant</src-type>
                  <value>1966/04/18</value>
                  <format>yyyy/MM/dd</format>
            </field>
            <field name = "COUNTRY_CODE" type='int'>
                  <src-type>constant</src-type>
                  <value>1</value>
            </field>
      </fields>
</mapping>
```

Attributes of “field”

Attribute	***Description***	***Required***
`name`	The name of the field, which will be used to identify the data elsewhere in the configuration file.	Yes
`type`	The internal data type of the field.	Yes
`pk`	Set to true/yes, defines this field as a primary key.	No; defaults to false/no.

Elements in “field”

Element Name	***Description***	***Required***
`src-type`	This determines the source type of this field. Possible source type are as follows. “***bi***” : This means that the field value is to be sourced from one of the fields in the set specified by the <***set-name***> element. “***constant***” : This means that the field value is the constant value specified by the <***value***> element. “***expression***” : This means that the field value contains an expression using $BV$ Bright Software value markers. Couple of possible use case are as follows. **Example 1** : An expression that can be used to specify a column value which will contain the next unique number generated by BrightIntegrator. **$BV$_NUN_$BV$** **Example 2** : An expression to concatenate two text fields into a single field. Note NAME and SURNAME are the existing fields in the data set. Also note the space between the fields. **$BV$NAME$BV$ $BV$SURNAME$BV$** **Example 3** : A numeric expression to calculate the total cost based on the quantity and price. Note that QTY and PRICE are the existing fields in the data set. Also note the multiplication char “*”. **$BV$QTY$BV$*$BV$SURNAME$BV$** **Example 4** : A numeric expression to calculate the profit based on the cost and sell prices. Note that COST and SELL_PRICE are the existing fields in	Yes.

	the data set. Also note the subtraction char "-". **$BV$SELL_PRICE$BV$-$BV$COST$BV$** **Example 5** : An expression that converts a field that contains temperature in degrees Celsius to degrees Fahrenheit. Note that CELSIUS is the existing field in the data set. **$BV$CELSIUS$BV$*1.8 + 32** Note that the available mathematical operations are +, -, *, /. Parenthesis (i.e '**(**' and '**)**') can be used. See Section "8.3.6.1 $BV$ Value Place Holder" for available place holder.	
set-name	Specifies the name of the set from which the field values is to be fetched.	Yes only if the src-type is "bi", otherwise not required
field-name	Specifies the name of the field to be fetched from the set specified using the "*set-name*" element. If this element is not specified, then the name of the transformation field (specified by the "*name*" attribute of the <*field*> element) will be assumed to the same with the field name to be read. By using this, the incoming field name can be changed to the name specified by the field name.	No Used only the src-type is bi
value	This is the element that contains the constant value.	Yes only if the src-type is "constant", otherwise not required
format	This specifies the format of the value specified by the <value> element.	No Only used with <value> element if the src-type is constant.
post-function	This element allows the execution of a specific function for the further transformation of the field value. This is available for all field types in the transformation mapping. This element, for instance, could be used to replace or to remove invalid characters from the field value before it is sent to BrightServer by using the example following functions:	No

	The following function will replace all of the degree special characters (“°”) with “DegreeC” string. **ReplaceChar, 176, “DegreeC”** The following function will remove the special degree character altogether from the field. **RemoveChar,176** See *Appendix G* for the list of available functions with syntax and examples.	

8.5 Queries

IMPORTANT NOTE: Use BrightBuilder's "Query→Export As Text" tool (mouse right click on the query name icon in BrightBuilder) to use queries created in BrightBuilder in your BrightIntegrator configuration file. This will eliminate manual creation of query objects for BrightIntegrator query definitions.

Queries are used by BrightServer™ and JDBC data sets to help define what data is to be transferred. The **queries** element contains one or more **query** elements. Below is a sample.

```
<query name="Query1">
   <tables>
      <table type="parent">ORDERS</table>
      <table type="child">ORDER_LINES</table>
   </tables>
   <relationships>
      <relationship>
         <source name="ORDERS" type="parent" multiplicity="one">
         <key>
            <column order="1">ORDER_NO</column>
         </key>
         </source>
         <source name="ORDER_LINES" type="child" multiplicity="many">
         <key>
            <column order="1">ORDER_NO</column>
         </key>
         </source>
      </relationship>
   </relationships>
   <condition operator="AND">
      <expression p="n">
         <table-name>ORDERS</table-name>
         <column-name>DEVICE_ID</column-name>
         <op>eq</op>
         <value type="string">
            <![CDATA[ 1224567  ]]>
         </value>
      </expression>
      <expression p="y">
         <parameter name="pDate">
            <desc />
         </parameter>
         <table-name>ORDERS</table-name>
         <column-name>SENT_DATE</column-name>
         <op>eq</op>
         <value type="dateTime" />
      </expression>
   </condition>
   <outputfields>
      <field>
         <table-name>ORDERS</table-name>
         <column-name>DEVICE_ID</column-name>
         <alias />
      </field>
      <field>
        <table-name>ORDERS</table-name>
        <column-name>ORDER_NO</column-name>
        <alias />
      </field>
      <field>
         <table-name>ORDERS</table-name>
         <column-name>LINE_COUNT</column-name>
         <alias />
      </field>
   </outputfields>
   <orderfields>
      <field>
        <table-name>ORDERS</table-name>
        <column-name>ORDER_NO</column-name>
        <orderby>asc</orderby>
      </field>
     </orderfields>
     <stored-procedure />
     <distinct-records>no</distinct-records>
     <online>no</online>
     <row-lock>no</row-lock>
  </query>
```

Attributes of "query"

Attribute	*Description*	*Required*
`name`	Name of the query. Data sets refer to this name when using this query.	Yes.

Query elements contain **tables**, **relationships**, **condition**, **outputfields**, and **orderfields** elements.

Elements in "query"

Element Name	*Description*	*Required*
`tables`	Specifies the tables within the query. Contains **table** elements, each with a **type** attribute, which may be "parent" or "child". The content of the element is the name of the table on the server.	Yes.
`relationships`	Specifies relationships between tables. There must be (n-1) relationships defined, where n is the number of tables in the query. Contains **relationship** elements. See below for details.	Yes, but only if there are multiple tables in the query
`condition`	Specifies the query condition. An empty element means no condition, therefore each record is returned. Contains **condition** elements. See below for details	Yes, but may be empty.
`outputfields`	Specifies the output fields for the query. Contains **field** elements.	No, if BrightServer™ is using the query, otherwise Yes.
`orderfields`	Specifies the order of appearance of the output fields. Contains **field** elements.	No.
`distinct-records`	Specifies if only distinct records are to be returned in the resultset. If yes, does not return duplicate records.	No, defaults to false/no.

`online`	Specifies if the query is online or not.	No, defaults to false/no.
`row-lock`	Specified if the records are to be row-locked when accessed by BrightIntegrator.	No, defaults to false/no.

Elements in “relationship”

Element Name	*Description*	*Required*
`source type='parent' multiplicity = 'one'`	The parent source for the relationship. Must have a **name** attribute, which corresponds to the table name. Contains one or more **key** elements.	Yes.
`source type = 'child' multiplicity = 'many'`	The child source for the relationship. Must have a **name** attribute, which corresponds to the table name. Contains one or more **key** elements.	Yes.

Elements in “key”

Element Name	*Description*	*Required*
`source type='parent' multiplicity = 'one'`	The parent source for the relationship. Must have a **name** attribute, which corresponds to the table name. Contains one or more **key** elements, containing one or more **column** elements. Each column has an **order** attribute, and names a table column.	Yes.
`source type = 'child' multiplicity = 'many'`	Similar to the parent source, but for the child source. Each column is matched between the parent and child, to form the relationship.	Yes.

Condition elements must contain an **expression** element. Otherwise a condition will have a comparison **operator** attribute, and multiple expressions. Possible comparison operators are “AND” and “OR. The expressions are compared using the comparison operator. A **condition** element may optionally contain another nested **condition** element.

Attributes of “expression”

Parameter	*Description*	*Required*
`p`	Tells whether the expression element contains a parameter or not. Possible values are “y” or “n”.	Yes.

Elements in "expression"

Element Name	*Description*	*Required*
`table-name`	Names the table that contains the value	Yes.
`column-name`	Names the table column that contains the value	Yes.
`op`	Operator for the expression. Possible values are lt, le, eq, ne, ge, gt, like.	Yes.
`parameter`	Must contain a **name** attribute, and may contain an optional **desc** (description) element.	Yes, but only if this expression contains a parameter.
`value`	Must have a **type** attribute. The content is the value to be used in conjunction with the expression operator.	Yes, but only if this expression does not contain a parameter.

Elements in "field"

Element Name	*Description*	*Required*
`table-name`	Names the table that contains the field	Yes.
`column-name`	Names the table column for the field	Yes.
`alias`	Optional alias for this field.	Yes, but only if this expression contains a function.
`function`	This element contains the SQL aggregate functions to be applied to the column. Supported values are count, sum, avg, min, or max.	No.
`orderby`	This element contains the distinct clause to be applied for ordering the query. Allowed values are yes and no.	No, but only used for **orderfields**.

8.5.1 User Defined Queries

In order to support JavaScript based user defined sync points on the server, we need to be able specify a special query type that we can use to send user defined payloads to the scripts executed on the server side. In many cases standard queries (XML query) or advanced SQL queries (basically a SQL SELECT statement) will be not adequate or will be restrictive in terms of specifying what it is that the remote device is trying to read from the script.

To provide maximum flexibility we will introduce a new query type where the user can define the format and the content of the query to be dispatched to server. This query data will be referred as "***payload***".

8.5.1.1 Query Spec XML Changes

```
<query name="MyQuery" version="5">
     <tables>...</tables>
     <is-user-query>yes</is-user-query>
     <user-query>
        <payload name="Payload"><![CDATA[my query data]]></payload>
     </user-query>
     <output-fields>…</output-fields>
</query>
```

is-user-query : Defines if the query is of a user query type or not.

user-query : This element contains the user query specific configuration elements.

payload : This defines the query payload together with the query parameter name.

output-fields : This is the existing element to contain the output fields from the parent table.

tables : This is the existing element to contain the name of the tables involved in the query.

8.6 Admin Element

BrightIntegrator configuration file allows you to configure system administration related options such as configuring the details of an email account to which an email is to be sent informing the sys admin of the result of a particular job execution. The following sections outline the available option in the admin section defined by the ***admin*** element in the configuration file.

```
<integrator version="2.0">
    <admin>
        <email-notification>
            <on-success>yes</on-success>
            <on-failure>yes</on-failure>
            <server>205.214.83.216</server>
            <port>25</port>
            <from>admin@mycompany.com.au</from>
            <to>admin@mycompany.com.au</to>
            <subject>BrightIntegrator job result</subject>
            <attach-file>c:/bi2/log/integrator.log</attach-file>
        </email-notification>
    </admin>
    <jobs version="2.0">    ...    </jobs>
    <tasks version='2.0'>    ...    </tasks>
    <queries>    ...    </queries>
    <data-sets>    ...    </data-sets>
    <mappings version = "2.0">    ...    </mappings>
</integrator>
```

Elements in "email-notification"

Element Name	*Description*	*Required*
`on-success`	If set to "yes" an email message will be sent if the job(s) are executed successfully (Valid values are "yes" or "no")	Yes
`on-failure`	If set to "yes" an email message will be sent if BrightIntegrator failed to execute job(s) (Valid values are "yes" or "no")	Yes
`server`	Email server address	Yes
`port`	Email server port number. This is set to default Port "25". Does not need to be specified.	No
`from`	Email address of the person who is sending the email.	Yes
`to`	Email address of the person to which the email is to be sent.	Yes
`subject`	Subject of the email	Yes
`attach-file`	Name of the file that can be sent with the email. This is optional.	No

8.7 Push Module

The Push Module configuration file defines the push mechanism to be executed by BrightIntegrator. This identifies the Publications, Subscriptions, Dispatchers, and Notifier elements of the Push Module. This also states what escalation actions are to be executed if a message fails to be sent etc.

The following is a sample Push Module configuration file. This file is referenced by the Push data-set.

```
<push-task version="1.0" def-version="1">
 <publications>
  <publication name="OrderItems" set="OrderItem">
   <field>OID</field>
  </publication>
 </publications>
 <subscriptions>
  <subscription publication="OrderItems">
   <subscriber type="file" file="/bi3/data/oidmap.csv">
    <subscriber-values>
     <value name="OID" type="int">1</value>
    </subscriber-values>
    <dispatcher-values dispatcher="FileDispatcher">
     <value name="OrderItem.file-name" type="string">2</value>
    </dispatcher-values>
   </subscriber>
  </subscription>
 </subscriptions>

 <dispatchers>
  <dispatcher name="FileDispatcher" destination="PushFile">
   <attribs/>
   <escalations>
    <escalation execute="on-failure">
     <destination name="EscalationFile">
      <value name="OrderItem.file-name" type="string">C:\OnFail.txt</value>
     </destination>
     <condition/>
     <write-values/>
    </escalation>
   </escalations>
  </dispatcher>
 </dispatchers>

 <notifier>
  <attribs>
   <value name="max-retries" type="int">10</value>
   <value name="retry-interval" type="int">300</value>
   <value name="delete-failed-jobs" type="boolean">no</value>
  </attribs>
  <message-store type="database">
   <value name="url"
type="string">jdbc:microsoft:sqlserver://bohr:1433;DatabaseName=bstest;Sele
ctMethod=cursor</value>
   <value name="jdbc-driver"
type="string">com.microsoft.jdbc.sqlserver.SQLServerDriver</value>
   <value name="username" type="string">bstest</value>
   <value name="password" type="string">bstest</value>
  </message-store>
 </notifier>
</push-task>
```

Elements in “push-task”

Element Name	*Description*	*Required*
`publications`	States the publications available to be used by the subscribers. Consist of more than one <publication> elements.	Yes
`subscriptions`	List the subsciption elements which defines the destination of the data to be sent via the dispatchers.	Yes
`dispatchers`	Defines the attributes and escalation actions of the dispatcher elements.	Yes
`Notifier`	Describes the notifier attributes and message store element.	Yes

The **publications** element consist of one or more **publication** elements.

Elements in “publications”

Element Name	*Description*	*Required*
`publication`		Yes

Attributes of “publication”

Element Name	*Description*	*Required*
`name`	Defines the name of the publication which will be used by the subscription element.	Yes
`set`	Specifies the set used by the publication. This will be the basis for the task data.	Yes

The **publication** element consist of one or more **field** elements. The field element references the data from the **set** specified in the publication element. The publication will then filter the **set** data based on the field element.

Elements in “publication”

Element Name	*Description*	*Required*
`field`	The field name to be used to filter the set data.	Yes

The **subscriptions** element consist of one or more **subscription** elements.

Elements in “subscriptions”

Element Name	*Description*	*Required*
`subsciption`	Defines the published data to be used by the subscription.	Yes

Attributes of "subscription"

Element Name	*Description*	*Required*
`publication`	The name of the publication that is referenced by the subscription.	Yes
`default-dispatcher`	The name of the default dispatcher.	No

The **subscription** element consist of one or more **subscriber** elements.

Elements in "subscription"

Element Name	*Description*	*Required*
`subsciber`	Defines the subcriber type. Can be based from a file that contains the list of subscriber details or based on a constant value.	Yes

Attributes of "subscriber"

Element Name	*Description*	*Required*
`type`	Type of the subscriber. Type can be a **constant** – which is the actual data values. It can also be a **file** – the values are indexing the file columns.	Yes
`file`	The name of the subscriber file.	Yes, if **type** is file.

The **subscriber** element consist of one or more **subscriber-values** elements.

Elements in "subscriber"

Element Name	*Description*	*Required*
`subsciber-value`	Defines the value of the data to be passed to the named fields in the referenced publication.	Yes
`dispatcher-values`	Defines the value of the attributes to be passed to the dispatcher elements for each subscriber.	Yes

Attributes of "dispatcher-values"

Element Name	*Description*	*Required*
`dispatcher`	The name of the dispatcher to be used by the subscriber.	Yes, if no **default-dispatcher** defined in the **subscription** element.

The **subscriber-values** element consist of one or more **value** elements which defines the data to be passed to the publication.

The **dispatcher-values** element consist of one or more **value** elements which defines the values to be passed to the attributes of the dispatcher. This is destination accessor specific. For example, if you are using a File accessor as a destination, you can change the value of the file-name for each subscriber. The following table details the dispatcher attributes that can be overwritten.

Destination	**Attribute**
File	SetName.file-name , append
BrightForms	server, port
BrightServer	url, username, password
Email	host, port, from, to, subject

The **dispatchers** element consist of one or more **dispatcher** elements.

Elements in "dispatchers"

Element Name	***Description***	***Required***
`dispatcher`	Defines the dispatcher name and destination	Yes

The **dispatchers** element consist of one or more **dispatcher** elements.

Elements in "dispatcher"

Element Name	***Description***	***Required***
`attribs`	Defines the attribute values to be passed to the dispatcher accessor.	No
`escalations`	List the escalations to be executed when sending the message to its destination	No

Attributes of "dispatcher"

Element Name	***Description***	***Required***
`name`	The name of the dispatcher referenced by the subscriber elements.	Yes
`destination`	Name of the destination data-set.	Yes

Elements in “escalations”

Element Name	*Description*	*Required*
`escalation`	Defines the escalation actions to be executed when sending the task data to the subscribers.	No

The **escalations** element consist of one or more **escalation** elements.

Elements in “escalation”

Element Name	*Description*	*Required*
`destination`	The destination name to be used by the escalation method.	Yes
`condition`	The condition or conditions for the escalation method.	No
`write-values`	Values to overwrite the set data fields. Can be used as a feedback mechanism to change a field depending on the status of the message sent.	No

If a write-value is not defined with the escalation action the values from the task data will be used for the write data. You can create a feedback mechanism within BrighIntegrator to change the value of the status of a record. For example, change the status of a JOB record if it was not sent successfully to the client devices.

Attributes of “escalation”

Element Name	*Description*	*Required*
`execute`	Defines when the escalation method will be executed. Possible values are: on-success, on-failure and always.	Yes

The “on-failure” escalation action will be executed if and only if the message status was failed. This means that a message is considered a failed message if the message was still unsuccesfully sent after the configured retries.

The **attribs**, **destination**, **condition** and **write-values** elements consists of one or more **value** element that defines the value to be passed to the mentioned elements. This element is discussed at the end of this section.

The **notifier** element contains the attributes for the notifier and the message store. The following are the attributes that you can change for the notifier element:

- ✓ **<max-retries>** - this is the maximum number that BrightIntegrator will retry sending a failed message.
- ✓ **<retry-interval>** - this is the time the message will be put into the “Idle” state. When the idle time elapses, BrightIntegrator will try resending the message depending on the maximum number of retries.
- ✓ **<delete-failed-jobs>** - a flag to automatically delete the failed jobs record from the message store.

The message store values are details of where the database of the message store is located, the following are the values that need to be specified:

- ✓ url
- ✓ jdbc-driver
- ✓ username
- ✓ password

Elements in "notifier"

Element Name	*Description*	*Required*
`attribs`	This defines the attributes of the notifier element such as maximum retries and idle interval time.	No
`message-store`	Describes the message store values used by the notifier.	No

Attributes of "message-store"

Element Name	*Description*	*Required*
`type`	Specifies the type of the message store. Type can either be "**database**" or "**memory**". By default message-store will be created in memory.	No

Note: If using a "memory" message-store when the computer system restarts, the message store will be reset.

The **value** element defines the name and type of the value to be passed to the following elements:

- Attribs
- Message-store
- Subscriber-values
- Dispatcher-values
- Destination of the escalation element
- Condition of the escalation element
- Write-values

The "value" element

Element Name	*Description*	*Required*
`value`	Defines the value name of the attribute.	Yes

Attributes of "value"

Element Name	*Description*	*Required*
`name`	Defines the name of the value to be used by the elements of the push module.	Yes
`type`	Specifies the data type.	Yes

8.8 Scheduler Component

The <schedules> element defines the scheduler component of the job processor. The scheduler can be based on a timer interval or a cron-trigger. See Appendix B for more details on cron-triggers and cron-expressions.

```
<schedules>
 <schedule name ="SimpleSchedule" type="simple">
  <value name="interval" type="int">300</value>
 </schedule>
 <schedule name ="CronSchedule" type="cron">
  <value name="cron-expression" type="string">0 0/30 8-17 26,27 * ?</value>
 </schedule>
</schedules>
```

Elements in "schedules"

Element Name	*Description*	*Required*
`schedule`	List the name of the scheduler components available in the configuration file.	Yes

Attributes of "schedule"

Element Name	*Description*	*Required*
`name`	The name of the scheduler component to be executed.	Yes
`type`	Defines the scheduler type. Can either be "simple" or "cron".	Yes

Elements in "schedule"

Element Name	*Description*	*Required*
`value`	Defines the value name of the scheduler type.	Yes

Attributes of "value"

Element Name	*Description*	*Required*
`name`	Defines the name of the value of the scheduler component. Can either be an "interval" or a "cron-expression".	Yes
`type`	Specifies the type of the value to be passed to the scheduler component. If using "interval" should pass an integer type. If using "cron-expression", the type should be string.	Yes

8.9 Data Value Formatting

Formatting can be used in files for fields that represent Boolean, numerical or date values.

Booleans can be formatted in several ways. All formats are case-insensitive. Booleans are recognised as “true” and “false”, “yes” and “no”, “0” and “1”. Alternatively, a customised format may be specified in the form “true-identifier/false-identifier”.

8.9.1 Number Formatting

Numbers can be formatted using the following pattern symbols.

Symbol	Location	Localized?	Meaning
`0`	Number	Yes	Digit
`#`	Number	Yes	Digit, zero shows as absent
`.`	Number	Yes	Decimal separator or monetary decimal separator
`-`	Number	Yes	Minus sign
`,`	Number	Yes	Grouping separator
`E`	Number	Yes	Separates mantissa and exponent in scientific notation. *Need not be quoted in prefix or suffix.*
`;`	Sub pattern boundary	Yes	Separates positive and negative sub patterns
`%`	Prefix or suffix	Yes	Multiply by 100 and show as percentage
`\u2030`	Prefix or suffix	Yes	Multiply by 1000 and show as per mille
¤ (`\u00A4`)	Prefix or suffix	No	Currency sign, replaced by currency symbol. If doubled, replaced by international currency symbol. If present in a pattern, the monetary decimal separator is used instead of the decimal separator.
`'`	Prefix or suffix	No	Used to quote special characters in a prefix or suffix, for example, `"'#'#"` formats 123 to `"#123"`. To create a single quote itself, use two in a row: `"# o''clock"`.

For example, to specify leading zeroes to pad out an integer to four digits, the pattern would be “0000”. Further, to specify exactly two decimal places, when given a double value, the pattern would be “#.00”. To specify two decimal places at the most, the pattern would be “#.##”.

More Examples

The following examples show how number patterns are interpreted for the given number: -123.45.

Number Pattern	Result
`#.00`	`-123.45`
`00000.00`	`-00123.45`
`%`	`-12345`

8.9.1.1 Number Alignment

In addition, the pattern may be prefixed with an *underscore*. This will cause the number to be right-aligned, within its field.

8.9.2 Date Formatting

Dates can be formatted using the following pattern symbols.

Letter	Date or Time Component	Presentation	Examples
`G`	Era designator	Text	`AD`
`y`	Year	Year	`1996`; `96`
`M`	Month in year	Month	`July`; `Jul`; `07`
`w`	Week in year	Number	`27`
`W`	Week in month	Number	`2`
`D`	Day in year	Number	`189`
`d`	Day in month	Number	`10`
`F`	Day of week in month	Number	`2`
`E`	Day in week	Text	`Tuesday`; `Tue`
`a`	Am/pm marker	Text	`PM`
`H`	Hour in day (0-23)	Number	`0`
`k`	Hour in day (1-24)	Number	`24`
`K`	Hour in am/pm (0-11)	Number	`0`
`h`	Hour in am/pm (1-12)	Number	`12`
`m`	Minute in hour	Number	`30`
`s`	Second in minute	Number	`55`
`S`	Millisecond	Number	`978`
`z`	Time zone	General time zone	`Pacific Standard Time`; `PST`; `GMT-08:00`
`Z`	Time zone	RFC 822 time zone	`-0800`

For example, to specify a short date with four digit year, the pattern would be “dd-MM-yyyy”. Further, to specify the 24 hour time, long date and four digit year, the pattern would be “HH:mm:ss EEE d MMM yyyy”.

More Examples

The following examples show how date and time patterns are interpreted for the given date and time: 2001-07-04 12:08.

Date and Time Pattern	Result
`h:mm`	`12:08`
`ddMMyy`	`040704`
`yyyyy.MMMMM.dd hh:mm`	`02004.July.04 12:08`

8.10 Logging Configuration

BrightIntegrator™ uses the popular Log4J framework for logging runtime information to the console, as well as to log files. The configuration file is log4j.xml, located in the conf directory. The log file “integrator.log” is located in the log directory.

There are five levels of logging information. They are DEBUG, INFO, WARN, ERROR, and FATAL. The DEBUG Level designates fine-grained informational events that are most useful to debug an application. The INFO level designates informational messages that highlight the progress of the application at coarse-grained level. The WARN level designates potentially harmful situations. The ERROR level designates error events that might still allow the application to continue running. The FATAL level designates very severe error events that will presumably lead the application to abort.

In order to change the level of logging, the user must edit the log4j.xml configuration file. Locate the “**root**” element, and in it, another element called “**priority**”. The “**value**” attribute sets the global level for logging. By default, this is set to INFO.

Note: Log4J is configured to send logging information to the console, and to files in the log directory. However, it applies a threshold on the console at the INFO level. This means that, even if DEBUG is configured, no DEBUG information will appear on the console. DEBUG information will appear in the log files as expected.

For more configuration information and examples see the Jakarta Log4j website: http://jakarta.apache.org/log4j

8.11 Scripts

User defined JavaScripts are defined using the “scripts” element.

```
<scripts>
   <script name="TestScript" type="file">C:\temp\MyScript.js</script>
</scripts>
```

Attributes of “script”

Attribute	***Description***	***Required***
Name	Name of the script	Yes.
Type	Type of script entry. “**file**” if the script is kept in an external file. Set always to “***file***”. The content of the <script> element contains the name of the physical file. (Note: “**embedded**” if the script is embedded the configuration file. This type is used only by BrightBuilder).	No (default “file”)

9.0 Last Run file

The Job Processor keeps track of the successful tasks executed in the Last Run file. This file is named ***last-run.xml***, and is by default located in the ***conf*** directory beneath where BrightIntegrator™ is run. The Last Run file is also used to store task-related state data as well.

Note that the location and name of the last run file can be set by using the [-l|--lastrun LASTRUN_FILE_NAME] command line option when running BrightIntegrator.

If BrightIntegrator™ experiences an error during a job, the next time it runs, the job will be resumed from the task that had not been successfully run. (This behaviour can be overridden using the –n command line option) This is achieved by the Job Processor writing to the Last Run file each time that a task was not completed. Once a job was run successfully, the Last Run for that job will be cleared.

Important: Ensure that each job has a different name for each configuration.

The Last Run file is also used for storing timestamps that are used to reading data from a BrightServer™ instance. Each time data is successfully read from BrightServer™, the timestamp associated with the read, is stored in the Last Run file. Then when the next read occurs, the timestamp in the Last Run file is retrieved and used as part of the read. In this way, only incremental data is returned by the read.

An example of the XML layout for the Last Run file is given below.

```
<lastrun version="2.0">
   <last-success>
      <job name="ProduceDifferenceFile">0</job>
      <job name="ExportBarCodes">0</job>
      <job name="EnterNewOrders">0</job>
   </last-success>
   <data-sets>
      <data-set name="ServerBarCodeTable">
      <value name="time-stamp">1104822837289</value>
      </data-set>
      <data-set name="BrightServerDebtorTable">
      <value name="time-stamp">0</value>
      </data-set>
      <data-set name="ServerTable1">
      <value name="time-stamp">0</value>
      </data-set>
   </data-sets>
</lastrun>
```

10.0 How Do I ?

10.1 JDBC and BrightServer import difference task

How do I create a task to get the difference between a JDBC and BrightServer source and save it back to BrightServer?

The task to generate the difference data set between the JDBC and BrightServer is shown below:

```
<task name="DiffImportBarCodes" >
  <source>ServerBarCodeTable</source>
  <old-source>BSBarcodeTable</old-source>
  <destination>BSBarcodeTable</destination>
  <description>
     <![CDATA[Calculate the difference, and save to BrightServer]]>
  </description>
</task>
```

The “ServerBarCodeTable” is defined as a JDBC data set and the BSBarcodeTable is defined as a BrightServer data set in the data-sets elements of the configuration file.

10.2 Exporting joined tables

How do I export customer and customer orders from BrightServer in one task but using two destination files?

Define the query with joined tables and simply specify the parent and child tables, and the relationships. See example below for CUST and CUST_ORDERS tables:

```
<queries>
  <query name="CustCOrdersQuery">
     <tables>
        <table type="parent">CUST</table>
        <table type="child">CUST_ORDERS</table>
     </tables>
     <relationships>
        <relationship>
              <source name="CUST" type="parent" multiplicity="one">
                <key>
                  <column order="1">CUST_NO</column>
                  <column order="2">DEL_CODE</column>
                </key>
              </source>
              <source name="CUST_ORDERS" type="child"
              multiplicity="many">
                <key>
                  <column order="1">CUST_NO</column>
                  <column order="2">DEL_CODE</column>
                </key>
              </source>
        </relationship>
     </relationships>
     <condition />
```

```
          <outputfields/>
          <orderfields/>
          <distinct-records>no</distinct-records>
          <online>no</online>
          <row-lock>no</row-lock>
    </query>
  </queries>
```

Then define the output files data-set as follows:

```
  <data-set name="OutFiles" type="File" >
   <sets>
    <set name="Cust">
          <file-name>c:/bi2/data/CUST.TXT</file-name>
          <mapping>CustCSVMapping</mapping>
      </set>
    <set name="CustOrders">
          <file-name>c:/bi2/data/CUST_ORDERS.TXT</file-name>
          <mapping>CustOrdersCSVMapping</mapping>
      </set>
   </sets>
  </data-set>
```

Appendix A – API (Pronto, Web Service) Configuration File

A1.0 Introduction

The API style data-set types (Pronto and Web Services) have their own dedicated XML configuration file. This file configures specific API calls to be made when data is received by the data-set. In simple terms this XML configuration file describes how the data needs to be processed by the external API provider. In the Pronto case, this is the Pronto Integration Engine (PIE), and in the Web Services case, this is the web server.

Both Pronto and Web Services data sets have in common the API module, which can handle consuming group data and submits it to the API server. Hence they also share the API configuration file.

The API configuration file defines which API to call together with the parameter details. The source of API parameters could be constant values (type ***constant***), values from the sets read by BrightIntegrator from other sources (type ***bi***), or the results of the API calls (type ***api***). The result parameters of each API call is cached by BrightIntegrator for subsequent API calls; however, BrightIntegrator will initially remove all the result parameters that the API call is supposed to return, thereby guaranteeing the correctness of the result parameter values that the subsequent API call rely on.

The document root element is **api-task** (also pronto-task is accepted for backwards compatibility). The top-level elements of the XML file are **pre-task**, **group**, **post-task**, and **apis**. An example of the each top-level element will be explained in the following sections.

API calls can be configured to be made at the following times:

- ❑ Before the task begins (pre-task)
- ❑ Before each data group (pre-group)
- ❑ For each record in a data group (group / set)
- ❑ After each data group (post-group)
- ❑ After the task ends (post-task)

A1.1 “pre-task” element

This optional element defines an API call to be made before the task begins. This is useful for initialisation purposes such as logging, creating connections or setting a state for the upcoming task. The value given for the **api** attribute should be defined in the **apis** element. The following example could be used to login before processing the group data,

```
    <pre-task api="login"/>
```

Attributes of “pre-task”

Attribute	*Description*	*Required*
`api`	The API method to be called before the task begins. The given value should refer to a corresponding entry in the **apis** element.	Yes

Important: The parameters of the pre-task API call should not reference **bi** value types as they are not available in this scope.

A1.2 “group” element

The **group** element forms the central part of the configuration file. A data group is a collection of records. Each record is associated to the group by pre-defined relationships. See section 2.3 Grouping Data for more details about grouping. An example of a data group would be a sales order, containing a single ORDER_HEADER record, along with multiple ORDER_ITEM records. The ORDER_HEADER record is said to be in the main set, and the ORDER_ITEM records would be members of a second set, in the group.

The core of the group element is the **sets** element, which defines an API call for each different set. So for the ORDER_HEADER main set, we might make the API call “createOrder”, and for the ORDER_ITEM second set, we might make the call “createLine”.

And then the **group** element would be,

```
<group>
        <sets>
            <set name="OrderHeader" api="createOrder"/>
            <set name="OrderItem" api="createLine"/>
        </sets>
</group>
```

The API calls are made in the order of appearance of the **set** elements.

Optional elements are **pre-group** and **post-group**. These elements declare apis that can be called before and after each group. To extend the above example, we may choose to use the **pre-group** element to log onto the server and then **post-group** element to submit the order so that we can complete the order for the group. Then the XML would become,

```
<group>
        <pre-group api="login"/>
        <sets>
            <set name="OrderHeader" api="createOrder"/>
            <set name="OrderItem" api="createLine"/>
        </sets>
        <post-group api=”submitOrder”/>
</group>
```

Note that pre-group and post group APIs will have access to the header record in the first set (main set) defined. In the above example that would be the order header record.

Also note that if a specified **bi** value does not exist in the set being submitted, then BrightIntegrator will search the value in the header record. By doing that, the user would not need to duplicate fields already in the header (main set) for the child sets.

Attributes of “group”

Attributes	*Description*	*Required*
`resend-failed`	Defines if the failed group has to be resent back to the server or not. If set to “false”, the failed groups will not be resent. If the attribute is not set, then the failed groups will be sent to the server. Default behaviour is to resend all failed groups.	No.

Elements in “group”

Element Name	*Description*	*Required*
`pre-group`	The API method to be called before the group is processed. The value given for the **api** attribute should be defined in the **apis** element.	No.
`sets`	Defines which API to be called for each set in the group. It contains one or more ***set*** elements.	Yes.
`post-group`	The API method to be called after the group is processed. The value given for the **api** attribute should be defined in the **apis** element.	No.

Attributes of “set”

Attributes	*Description*	*Required*
`name`	The name of the set. This name references the set name that was given in the original source data-set from where the data was read.	Yes.
`api`	The API method to be called for each record in the set. The value given for the **api** attribute should be defined in the **apis** element.	Yes.

A1.3 "post-task" element

This optional element defines an API call to be made after the task ends. This is useful for tearing down connections or restoring the state after the task. The value given for the **api** attribute should be defined in the **apis** element. The following example could be used to login before processing the group data,

```
    <post-task api="logoff"/>
```

Attributes of "post-task"

Attribute	*Description*	*Required*
`api`	The API method to be called after the task ends. The given value should refer to a corresponding entry in the **apis** element.	Yes

A1.4 "apis" element

The **apis** element defines the API calls that are declared in the other elements. The apis element contains one or more api elements.

Each **api** element has a **name** attribute, which is used as its reference throughout this file. The **external-name** element (pronto-name is also accepted for backwards compatibility) defines the actual API method name as the server knows it. If an error is returned from the server, then there is an optional **on-error** attribute which can be used to make another API call. Any errors that occur from **on-error** calls are ignored, so that the potential for infinite loops is avoided.

The api element can also optionally write return fields to a csv file as a form of feedback mechanism from the connection.

```
<api name="createOrder" on-error="">
     <on-error>
          <!—on-error details -->
     </on-error>
     <external-name>create-so</external-name>
     <params>
          <!—params details -->
     </params>
     <results>
          <!—results details -->
     </results>
     <file-feedbacks>
          <!—file-feedbacks details -->
     </file-feedbacks>
</api>
```

There is also an optional **on-error** element that allows BrightIntegrator to define multiple on-error API calls. It has the following format:

```
<on-error>
     <api>CancelOrder</api>
     <api>SendEmail</api>
</on-error>
```

If there is a single on-error API that needs to be called, the "on-error" attribute can be used. On the other hand, you do not need to use the "on-error" attribute, since the on-error element can also be used to define a single on-error API. If both on-errors are used, BrightIntegrator will call the on-error API specified by the "on-error" attribute and all the on-error APIs defined by the on-error element. As an example,

```
<api name="createOrder" on-error="CancelOrder">
     <on-error>
          <api>SendEmail</api>
     </on-error>
     …
</api>
```

is equal to:

```
<api name="createOrder" on-error="">
     <on-error>
          <api>CancelOrder</api>
          <api>SendEmail</api>
     </on-error>
     …
</api>
```

Attributes of "api"

Attributes	***Description***	***Required***
`name`	The name of the api. This name references this api throughout the rest of the file.	Yes
`on-error`	The API method to be called if an error occurs during the call of this api.	No

Elements in "api"

Element Name	***Description***	***Required***
`on-error`	Allows multiple on-error API names to be defined.	No
`external-name`	The actual API method name according to the server.	Yes
`params`	Defines the input parameters to give to the API call.	No
`results`	Defines the output results from the API call.	No

The **params** element contains one or more **param** elements. Each **param** element corresponds to an input parameter for the API call. An input parameter must be one of three types, constant – an explicitly defined fixed value, bi – a value taken from a named field in the input set data, or api – a value taken from a named result of a previous API call.

```
<params>
      <param name="lp-auth-user-number">
            <src-type>api</src-type>
            <src-name>lr-user-number</src-name>
      </param>
      <param name="lp-auth-accountcode">
            <src-type>constant</src-type>
            <value>123456</value>
      </param>
</params>
```

Attributes of “param”

Attributes	*Description*	*Required*
`name`	The name of the input parameter, according to the API call.	Yes.

Elements in “param”

Element Name	*Description*	*Required*
`src-type`	Valid values are: ***constant***, ***api***, ***bi***, ***systemtime, set, taskdata*** If the src-type is ***systemtime***, BrightIntegrator will provide the current system date-time at the time of API call. If the src-type is ***set*** or ***taskdata***, BrightIntegrator will provide the XML string in the Bright XML format, for either just the set or the whole taskdata, respectively. See Appendix F for details.	Yes.
`src-name`	If **src-type** is ***bi***: then **src-name** is the name of a field in the input data set. If **src-type** is ***api***: then **src-name** is the name of a result from a previous API call.	Only if src-type is ***api*** or ***bi***.
`format`	The format of the field. See section 8.9 Data Value Formatting	No
`value`	A fixed value.	Only if src-type is ***constant***.

The **results** element contains one or more **result** elements. One result **element** is required to be defined for each output that we wish to inspect the value of. Each **result** element contains one or more case **elements**, which specify value comparisons. The result value is compared with each case down the list until the first match is found. Then the instructions attached to the matching case are followed. A case may specify that the on-error API call is to be made. Following this the case command is carried out.

Possible case commands are:
"**abort-task**": The current task is aborted and any further processing is cancelled.
"**exit-task**": The current task processing is cancelled, but any post-processing is still carried out.
"**abort-group**": The current group processing is aborted, and its post-processing is cancelled, and the next group will be processed.
"**exit-group**": The current group processing is cancelled, but its post-processing is still carried out, and the next group will be processed.
"**continue**": Processing continues uninterrupted, but the on-error action is invoked. This command is used as a placeholder if we just want the on-error action to occur.
"**repeat**": Processing continues uninterrupted, and the current API call will be invoked again. This is used for API calls which can return data iteratively.

Possible case operators are:

Operator	***Meaning***
eq	equal to
ne	not equal to
lt	less than
le	less than or equal to
gt	greater than
ge	greater than or equal to

For instance, say the result value returned from the API call is -1. Then a case with operator=”eq” and code=”0” will not match, because -1 is not equal to 0. A case with operator=”lt” and code=”0” will match, because -1 is less than 0.

```
<results>
      <result name="lr-result-status" type="int">
            <case command="abort-task" run-on-error="no">
                  <operator>ne</operator>
                  <code>0</code>
            </case>
      </result>
</results>
```

Attributes of “result”

Attributes	***Description***	***Required***
name	The name of the output result, according to the API.	Yes.
type	The data type of the result. Internal data types are accepted.	Yes.

The **result** element contains one or more **case** elements.

Attributes of “case”

Attributes	*Description*	*Required*
`run-on-error`	Set to “yes” or “true” if the on-error api is to be called in the case matches. This will occur before command.	No.
`command`	The command to be carried out if the case matches. Possible values are: abort-task, exit-task, abort-group, exit-group, continue	Yes.

Elements in “case”

Element Name	*Description*	*Required*
`operator`	The comparison operator. Possible values are: eq, lt, le, gt, ge, ne.	Yes.
`code`	A fixed value to be used as a part of the comparison.	No.

The **file-feedbacks** element is an optional element that allows BrightIntegrator to write the return output from Pronto to multiple feedback files per API call. The file-feedbacks element contains one or more **file-feedback** elements. One file-feedback element is required to be defined for each output file.

```
<file-feedbacks>
   <file-feedback name="c:\bi2\Feedback.txt" type="csv" append="no">
      <file-field>
         <src-type>bi</src-type>
         <src-name>OID</src-name>
      </file-field>
      <file-field>
        <src-type>api</src-type>
        <src-name>lr-sales-order-number</src-name>
      </file-field>
      <file-field>
         <src-type>constant</src-type>
         <value>hello</value>
      </file-field>
      <file-field>
         <src-type>bi</src-type>
         <src-name>dtOrdered</src-name>
         <format>dd-MM-yyyy</format>
      </file-field>
   </file-feedback>
</file-feedbacks>
```

Elements in “file-feedbacks”

Element Name	*Description*	*Required*
`file-feedback`	Defines the output file.	Yes

Attributes of “file-feedback”

Attributes	*Description*	*Required*
`name`	The name of the output file.	Yes
`type`	The file type should always be a comma-separated file type.	Yes
`append`	If set to yes, then when writing to this data-set, information will be appended, if the file already exists. Otherwise the file will be rewritten from the beginning.	No; defaults to “yes”
`on-success`	If set to yes, when the api is executed successfully, the feedback-file will be written on. Otherwise the feedback-file write operation will be ignored. If this attribute does not exist in the configuration file, BrightIntegrator will still write to the feedback-file on api success.	No; defaults to “yes”
`on-failure`	If set to yes, when the api fails the feedback-file will be written on; otherwise the feedback-file write operation will be ignored. If this attribute does not exist in the configuration file, BrightIntegrator will still write to the feedback-file on api failure.	No; defaults to “yes”

The **file-feedback** element contains one or more **file-field** elements. One file-field element is required to be defined for each return output that is required to write to the output file.

Elements in “file-field”

Element Name	*Description*	*Required*
`src-type`	Valid values are: ***constant***, ***api***, ***bi***, ***systemtime*** If the src-type is ***systemtime***, BrightIntegrator will return the current system date-time at the time of writing the file field value to the feedback file.	Yes.
`src-name`	If **src-type** is ***bi***: then **src-name** is the name of a field in the input data set. If **src-type** is ***api***: then **src-name** is the name of a result from a previous API call.	Only if src-type is ***api*** or ***bi***.
`format`	The format of the field. See section 8.9 Data Value Formatting	No
`value`	A user defined value to be written to the file instead of the null value returned by the output result source.	Only if src-type is ***constant***.

Appendix B – Cron Expressions

This section explains what Cron and Cron expressions are. There is also a list of examples that can be used within your scheduler component.

B1.0 Cron Definition

Cron is the name of program that enables unix users to execute commands or scripts (groups of commands) automatically at a specified time/date. This allows users to create a CronTrigger that can fire a job schedule that recurs based on calendar-like notations such as “At 2:00 pm every last Friday of the month” or “Every 8:00am and 9:00am every Monday to Friday, rather than specified intervals.

B1.1 Cron Expressions

A cron expression is a string comprised of 6 or 7 fields separated by white space which defines your CronTrigger and describes individual details of the schedule. The 6 mandatory and 1 optional fields are as follows:

Field Name	Allowed Values	Allowed Special Characters
Seconds	0-59	, - * /
Minutes	0-59	, - * /
Hours	0-23	, - * /
Day-of-month	1-31	, - * ? / L W C
Month	1-12 or JAN_DEC	, - * /
Day-of-week	1-7 or SUN-SAT	, - * ? / L C #
Year (Optional)	Empty, 1970-2099	, - * /

Special Character Definition:

Special Character	Description
*	(Asterisk) Specifies all values. For example, “*” in the minute field means every minute.
?	(Question mark) This is used to specify “no specific value” in the day-of-month and day-of-week fields. This is useful when you need to specify something in one of the two fields but not the other.
-	(Dash) Defines a range. For example, “10-12” in the hour field means “the hours 10,11 and 12”.
,	(Comma) Used to specify additional values. For examples, "MON,WED,FRI" in the day-of-week field means "the days Monday, Wednesday, and Friday".
/	(Forward slash) Used to specify increments. For example "0/15" in the seconds field means "the seconds 0, 15, 30, and 45". And "5/15" in the seconds field means "the seconds 5, 20, 35, and 50". You can also specify '/' after the '*' character - in this case '*'

	is equivalent to having '0' before the '/'.
L	This character is short-hand for "last" and is only allowed for the day-of-month and day-of-week fields. For example, the value "L" in the day-of-month field means "the last day of the month" - day 31 for January, day 28 for February on non-leap years. If used in the day-of-week field by itself, it simply means "7" or "SAT". But if used in the day-of-week field after another value, it means "the last xxx day of the month" - for example "6L" means "the last friday of the month". When using the 'L' option, it is important not to specify lists, or ranges of values, as you'll get confusing results.
W	This character is used to specify the weekday (Monday-Friday) nearest the given day. It is only allowed for the day-of-month field. For example, if you were to specify "15W" as the value for the day-of-month field, the meaning is: "the nearest weekday to the 15th of the month". So if the 15th is a Saturday, the trigger will fire on Friday the 14th. If the 15th is a Sunday, the trigger will fire on Monday the 16th. If the 15th is a Tuesday, then it will fire on Tuesday the 15th. However if you specify "1W" as the value for day-of-month, and the 1st is a Saturday, the trigger will fire on Monday the 3rd, as it will not 'jump' over the boundary of a month's days. The 'W' character can only be specified when the day-of-month is a single day, not a range or list of days.
LW	Translates to “last weekday of the month”. This combination can be used for the ay-of-month field.
C	This character is short-hand for "calendar" and is only allowed for the day-of-month and day-of-week fields. This means values are calculated against the associated calendar, if any. If no calendar is associated, then it is equivalent to having an all-inclusive calendar. A value of "5C" in the day-of-month field means "the first day included by the calendar on or after the 5th". A value of "1C" in the day-of-week field means "the first day included by the calendar on or after sunday".
#	This character is used to specify "the nth" XXX day of the month and is only allowed for the day-of-week field. For example, the value of "6#3" in the day-of-week field means the third Friday of the month (day 6 = Friday and "#3" = the 3rd one in the month). Other examples: "2#1" = the first Monday of the month and "4#5" = the fifth Wednesday of the month. Note that if you specify "#5" and there is not 5 of the given day-of-week in the month, then no firing will occur that month.

NOTE: The legal characters and the names of months and days of the week are not case sensitive.

Here are some examples:

Expression	**Meaning**
"0 0 12 * * ?"	Fire at 12pm (noon) every day
"0 15 10 ? * *"	Fire at 10:15am every day
"0 15 10 * * ?"	Fire at 10:15am every day
"0 15 10 * * ? *"	Fire at 10:15am every day
"0 15 10 * * ? 2005"	Fire at 10:15am every day during the year 2005
"0 * 14 * * ?"	Fire every minute starting at 2pm and ending at 2:59pm, every day

"0 0/5 14 * * ?"	Fire every 5 minutes starting at 2pm and ending at 2:55pm, every day
"0 0/5 14,18 * * ?"	Fire every 5 minutes starting at 2pm and ending at 2:55pm, AND fire every 5 minutes starting at 6pm and ending at 6:55pm, every day
"0 0-5 14 * * ?"	Fire every minute starting at 2pm and ending at 2:05pm, every day
"0 10,44 14 ? 3 WED"	Fire at 2:10pm and at 2:44pm every Wednesday in the month of March.
"0 15 10 ? * MON-FRI"	Fire at 10:15am every Monday, Tuesday, Wednesday, Thursday and Friday
"0 15 10 15 * ?"	Fire at 10:15am on the 15th day of every month
"0 15 10 L * ?"	Fire at 10:15am on the last day of every month
"0 15 10 ? * 6L"	Fire at 10:15am on the last Friday of every month
"0 15 10 ? * 6L"	Fire at 10:15am on the last Friday of every month
"0 15 10 ? * 6L 2002-2005"	Fire at 10:15am on every last friday of every month during the years 2002, 2003, 2004 and 2005
"0 15 10 ? * 6#3"	Fire at 10:15am on the third Friday of every month
"0 0/5 * * * ?"	Fires every 5 minutes
"10 0/5 * * * ?"	Fires every 5 minutes, at 10 seconds after the minute (i.e. 10:00:10 am, 10:05:10 am, etc.).
"0 30 10-13 ? * WED,FRI"	Fires at 10:30, 11:30, 12:30, and 13:30, on every Wednesday and Friday.
"0 0/30 8-9 5,20 * ?"	Fires every half hour between the hours of 8 am and 10 am on the 5th and 20th of every month. Note that the trigger will NOT fire at 10:00 am, just at 8:00, 8:30, 9:00 and 9:30.

Some scheduling requirements are too complicated to express with a single trigger - such as "every 5 minutes between 9:00 am and 10:00 am, and every 20 minutes between 1:00 pm and 10:00 pm". The solution in this scenario is to simply create two triggers, and register both of them to run the same job.

References:
http://quartz.sourceforge.net/javadoc/org/quartz/CronTrigger.html
http://www.opensymphony.com/quartz/wikidocs/TutorialLesson6.html

Appendix C – Running BrightIntegrator as a Windows Service

The “service” sub directory contains all the necessary files for installing BrightIntegrator as a Windows service.

Important to note that BrightIntegrator can only be installed and run as a Windows Service if it has configured schedules to run continuously otherwise it will run the jobs configured once and exit.

When running as a service, BrightIntegrator needs to point to a BrightIntegrator XML configuration file. This is configured in the "wrapper.conf" file located in the “*service*” sub directory where the BrightIntegrator is installed.

By default it refers to the BrightIntegrator configuration file "config.xml" located in the "conf" directory (i.e. ..\conf\config.xml).

If you are using a configuration file with the same name in the same location, then you do not need to change the "wrapper.conf" file, and you can skip this step.

If you are using a configuration file that has a different name and/or is located somewhere else, then go to Line 54 of the "wrapper.conf" file and edit the following line.

```
wrapper.app.parameter.2=-c ..\conf\config.xml
```

You can change this line as required. For instance to :

```
wrapper.app.parameter.2=-c ..\new_location\myconfigfile.xml
```

NOTE: When BrightIntegrator is installed by the InstallShield based setup program, then a Windows Service named “BrightIntegrator” will also be installed by the setup program. The status of the BrightIntegrator service installed is ***manual***. After configuring wrapper.conf file as described above, go to *Services* in the Windows Control Panel, and set the status to “Automatic” and start the BrightIntegrator service to run BrightIngerator.

In order to install the BrightIntegrator service manually, follow the following instructions.

To install BrightIntegrator as a Windows service, run

InstallBI3Service.bat

To remove BrightIntegrator as a Windows service, run

UninstallBI3Service.bat

Troubleshooting

To run BrightServer, you must configure necessary schedules for the jobs in question. Otherwise BrightIntegrator will run once and exit the service.

If you have problems when trying to start BrightIntegrator as a Windows service, you can use RunBI3.bat to run BrightIntegrator in a command line interface and check the errors and fix the configuration issues that is preventing BrightIntegrator from running as a service.

You can also check out the "wrapper.log" file in the log directory (..\log).

Appendix D – How to connect BrightServer via a secure connection using “truststore”

In order to connect to BrightServer via a secure https port, the file shipped in the root BrightIntegrator directory must be copied to the home directory of the user running BrightIntegrator. The name of the file is “***truststore***” and it contains the necessary digital certificates signed by Bright Software in order to communicate with BrightServer via the dedicated https port. This port is by default configured to be on port 8443.

For example, on Windows platform, if the account name of the user who is running BrightIntegrator is *jsmith*, then the file ***truststore*** needs to be copied “*C:\Documents and Settings\jsmith*”.

Important to note that if BrightIntegrator is configured to be running as a Windows service, then the file ***truststore*** must be copied to the home directory of the user account which is configured to run BrightIntegrator as the service. This can be configured via the ‘Log On’ tab of the BrightIntegrator service entry created.

Appendix E – Formatting Objects

When an XML File Data Set is configured to apply an XSL transform, it is possible to also configure BrightIntegrator to interpret the XSL output as a Formatting Object tree. In this case, BrightIntegrator is able to render the resulting pages in a specified format, to the File Data Set filename.

Output formats currently supported include PDF, PCL, PS, SVG, MIF, TXT, and printing directly to the default printer.

BrightIntegrator internally uses Apache FOP (Formatting Objects Processor)., which is a partial implementation of the XSL-FO Version 1.0 W3C Recommendation.

Support for each of the XSL-FO standard objects and properties are detailed in FOP Compliance on the Apache site, http://xmlgraphics.apache.org/fop/compliance.html.

Appendix F – TaskData XML Object

The following XML format is used to represent a TaskData data object.

```
<data>
  <table name="myTable">
    <columns>
      <col type="string">Name</col>
      <col type="string">Address</col>
      <col type="int">Age</col>
      <col type="dateTime">dt_birth</col>
      <col type="float">salary</col>
      <col type="double">target</col>
      <col type="boolean">IsManager</col>
    </columns>
    <records>
      <record>
         <item>John</item>
         <item>Sydney 2000</item>
         <item>20</item>
         <item>1984-12-11T11:20:00.000Z</item>
         <item>20000.0</item>
         <item>220000.0</item>
         <item nil="true"/>
      </record>
      <record>
         <item>Helga</item>
         <item nil="true"/>
         <item>18</item>
         <item>1986-01-01T11:20:00.000Z</item>
         <item>18000.3</item>
         <item>220000.0</item>
         <item>false</item>
      </record>
    </records>
  </table>
  <table name=...>
  ...
  </table>
  <table name=...>
  ...
  </table>
  ...
</data>
```

Appendix G – Transformation Field Functions

The following table contains the available functions that can be used in the transformation field mappings.

Function	Description/Syntax
If	**Description:** Evaluates a conditional expression, and returns one of two possible values. **Syntax:** If, *v1*, *op*, *v2*, *true-value*, *false-value* where *v1*: Value to be used in the conditional expression *op*: Comparison operator for the conditional expression *v2*: Value to be used in the conditional expression *true-value*: Returns this value if the condition is true *false-value*: Returns this value if the condition is false **NOTE:** The comparison operator must be one of the following: “lt”: less than. “le”: less than or equal to. “eq”: equal to. “ne”: not equal to. “ge”: greater than or equal to. “gt”: greater than. **Example:** **If,$BV$FIELD1$BV$,eq,"str",”they_match","NO_MATCH"** If the value for FIELD1 in this record is equal to “str” then return the value “they match”, otherwise return the value “NO_MATCH”. **Example:** **If, $BV$FIELD_DOUBLE$BV$, lt, 333.0, 0.0, 1.1** If the value for FIELD_DOUBLE in this record is less than 333.0 then return the value 0.0, otherwise return the value 1.1.
Replace	**Description:** Replaces all occurrences of one string with another in the field. **Syntax: Replace,*s1*, *s2*** where *s1* : String to be searched *s1* : New string to replace all occurrences of s1 **Example:** **Replace,”abc”,”xxx”** If the field value were “abcdefabc”, after the execution of the function, the field value would contain “xxxdefxxx”
ReplaceChar	**Description:** Replaces all occurrences of a character in the field with the string specified **Syntax: ReplaceChar,*ddd*, *str*** where *ddd* : Decimal value of the code of the Unicode character to be searched *str* : New string to replace all occurrences of the character **Example:**

	Replace,176,"DegreeC" If the field value were "37 °", after the execution of the function, the field value would contain "37 DegreeC" **IMPORTANT NOTE** : The character code must be the Unicode character's decimal value. Please use a Unicode Character Map to find the decimal value that needs to be used.
ReplaceNonPrintable	**Description:** Replaces all occurrences of non-printable characters in the field with the character specified. **Syntax: ReplaceNonPrintable,** *str* where *str* : Replacement character to be used to replace all non-printable characters. (Note: that the replacement character is still specified as a String, however only the first character will be used) **Example:** **ReplaceNonPrintable,"?"** If the field value contains non-printable characters such as "0x0002Normal0x0000Text0x008A", after the execution of the function, the field value would contain "?Normal?Text?". **NOTE**: The precise algorithm is to be replace all ASCII characters less than 32, and all ASCII characters greater than 126, with the exceptions of 0x0009 (tab), 0x000A (LF) and 0x000D (CR).

Remove	**Description:** Removes all occurrences of a string from the field. **Syntax** : **Remove,***str* where *str* : Substring to be searched and removed **Example:** **Remove,"$"** If the field value were "$abc$def$", after the execution of the function, the field value would contain "abcdef"
RemoveChar	**Description:** Removes all occurrences of a character from the field. **Syntax** : **RemoveChar,***ddd* where *ddd* : Decimal value of the code of the Unicode character to be removed from the field value **Example:** **RemoveChar,176** If the field value were "37°", after the execution of the function, the field value would contain "37" See the note above regarding the Unicode character value.
ToUpper	**Description:** Converts all the characters in the field to uppercase **Syntax: ToUpper** **Example:** **ToUpper** If the field value were "hello world", after the execution of the function, the field value would contain "HELLO WORLD"
ToLower	**Description:** Converts all the characters in the field to lowercase **Syntax: ToLower**

	Example: **ToLower** If the field value were "HELLO WORLD", after the execution of the function, the field value would contain "hello world"
Mid	**Description:** Extracts the number of characters specified starting from the location requested **Syntax: Mid,*start*,*count*** where *start* : Starting position *count* : Number of characters to be extracted **Example:** **Mid,5,3** If the field value were "0123456789", after the execution of the function, the field value would contain "567"
Left	**Description:** Extracts the first (leftmost) number of characters specified from the field **Syntax: Left,*count*** where *count* : Number of characters to be extracted **Example:** **Left,3** If the field value were "0123456789", after the execution of the function, the field value would contain "012"
Right	**Description:** Extracts the last (rightmost) number of characters specified from the field **Syntax: Left,*count*** where *count* : Number of characters to be extracted **Example:** **Right,3** If the field value were "0123456789", after the execution of the function, the field value would contain "789"